User's Manual

Jean J. Labrosse

Micriμm Press
Weston, FL 33326

µC/OS-III User's Manual

1. µC/OS-III User Manual 4
1.1 Preface 5
1.2 Introduction 13
1.2.1 Foreground-Background Systems 15
1.2.2 Real-Time Kernels 16
1.2.3 RTOS Real-Time Operating System 19
1.2.4 uC-OS-III 20
1.2.5 uC-OS uC-OS-II and uC-OS-III Features Comparison 25
1.2.6 uC-Probe 28
1.2.7 Documentation Conventions 29
1.3 Directories and Files 30
1.3.1 Application Code 34
1.3.2 uC-OS-III Board Support Package BSP 36
1.3.3 uC-OS-III CPU Independent Source Code 38
1.3.4 uC-OS-III CPU Specific Source Code 42
1.3.5 uC-CPU CPU Specific Source Code 44
1.3.6 uC-LIB Portable Library Functions 47
1.4 Getting Started with uC-OS-III 50
1.4.1 Single Task Application 51
1.4.2 Multiple Tasks Application with Kernel Objects 58
1.5 Critical Sections 65
1.5.1 Disabling Interrupts 66
1.5.1.1 Measuring Interrupt Disable Time 67
1.5.2 Locking the Scheduler 68
1.5.2.1 Measuring Scheduler Lock Time 69
1.5.3 uC-OS-III Features with Longer Critical Sections 70
1.6 About Task Management 71
1.6.1 Assigning Task Priorities 79
1.6.2 Determining the Size of a Stack 81
1.6.3 Detecting Task Stack Overflows 82
1.6.4 Task Management Services 88
1.6.5 Task Management Internals 89
1.6.6 Internal Tasks 109
1.6.6.1 The Idle Task 110
1.6.6.2 The Tick Task 112
1.6.6.3 The Statistic Task 119
1.6.6.4 The Timer Task 123
1.6.6.5 The ISR Handler Task 125
1.7 The Ready List 126
1.7.1 Priority Levels 127
1.7.2 About the Ready List 131
1.7.3 Adding Tasks to the Ready List 134

1.8 Scheduling . . . 136
1.8.1 Preemptive Scheduling . . . 137
1.8.2 Scheduling Points . . . 140
1.8.3 Round-Robin Scheduling . . . 142
1.8.4 Scheduling Internals . . . 144
1.9 Context Switching . . . 147
1.9.1 OSCtxSw . . . 150
1.9.2 OSIntCtxSw . . . 153
1.10 Interrupt Management . . . 155
1.10.1 Handling CPU Interrupts . . . 156
1.10.2 Typical uC-OS-III Interrupt Service Routine . . . 158
1.10.3 FAST Interrupt Service Routine . . . 161
1.10.4 Processors with Multiple Interrupt Priorities . . . 162
1.10.5 All Interrupts Vector to a Common Location . . . 165
1.10.6 Every Interrupt Vectors to a Unique Location . . . 167
1.10.7 Direct and Deferred Post Methods . . . 168
1.10.8 Direct vs. Deferred Post Method . . . 174
1.10.9 The Clock Tick . . . 176
1.11 Pend Lists . . . 179
1.12 Time Management . . . 185
1.13 Timer Management . . . 194
1.13.1 Timer Management Internals . . . 198
1.14 Resource Management . . . 207
1.14.1 Disable-Enable Interrupts . . . 210
1.14.2 Lock-Unlock . . . 212
1.14.3 Semaphores in uC-OS-III . . . 213
1.14.3.1 Binary Semaphores . . . 215
1.14.3.2 Counting Semaphores . . . 222
1.14.3.3 Notes on Semaphores . . . 224
1.14.3.4 Semaphore Internals . . . 225
1.14.3.5 Priority Inversions . . . 230
1.14.4 Mutual Exclusion Semaphores . . . 232
1.14.4.1 Mutual Exclusion Semaphore Internals . . . 237
1.14.5 Should You Use a Semaphore Instead of a Mutex . . . 243
1.14.6 Deadlocks . . . 244
1.15 Synchronization . . . 247
1.15.1 Semaphores . . . 248
1.15.2 Task Semaphore . . . 262
1.15.3 Event Flags . . . 268
1.15.4 Synchronizing Multiple Tasks . . . 281
1.16 Message Passing . . . 283
1.16.1 Messages . . . 284
1.16.2 Message Queues . . . 285

1.16.3 Task Message Queue 287
1.16.4 Bilateral Rendez-vous 288
1.16.5 Flow Control 290
1.16.6 Keeping the Data in Scope 292
1.16.7 Using Message Queues 294
1.16.8 Clients and Servers 302
1.16.9 Message Queues Internals 303
1.17 Pending on Multiple Objects 306
1.18 Memory Management 314
1.18.1 Creating a Memory Partition 316
1.18.2 Getting a Memory Block from a Partition 320
1.18.3 Returning a Memory Block to a Partition 321
1.18.4 Using Memory Partitions 322
1.19 Porting uC-OS-III 326
1.19.1 Conventions 331
1.19.2 uC-CPU 332
1.19.3 uC-OS-III Port 341
1.19.4 Board Support Package 366
1.19.5 Testing a Port 368
1.20 Run-Time Statistics 383
1.20.1 General Statistics Run-Time 384
1.20.2 Per-Task Statistics Run-Time 390
1.20.3 Kernel Object Run-Time 395
1.20.4 os_dbg.c Static 399
1.20.5 os_cfg_app.c Static 410

µC/OS-III User Manual

µC/OS-III is a highly portable, ROMable, scalable, preemptive, real-time, deterministic, multitasking kernel for microprocessors, microcontrollers and DSPs.

Copyright 2015 Micrium Inc.

Preface

What is µC/OS-III?

µC/OS-III (pronounced “Micro C O S Three) is a scalable, ROMable, preemptive real-time kernel that manages an unlimited number of tasks. µC/OS-III is a third-generation kernel and offers all of the services expected from a modern real-time kernel, such as resource management, synchronization, inter-task communications, and more. However, µC/OS-III offers many unique features not found in other real-time kernels, such as the ability to complete performance measurements at run-time, to directly signal or send messages to tasks, achieve pending on multiple kernel objects, and more.

Why a new µC/OS version?

The µC/OS series, first introduced in 1992, has undergone a number of changes over the years based on feedback from thousands of people using and deploying its evolving versions.

µC/OS-III is the sum of this feedback and experience. Rarely used µC/OS-II features were eliminated and newer, more efficient features and services, were added. Probably the most common request was to add round robin scheduling, which was not possible for µC/OS-II, but is now a feature of µC/OS-III.

µC/OS-III also provides additional features that better exploit the capabilities of today’s newer processors. Specifically, µC/OS-III was designed with 32-bit processors in mind, although it certainly works well with 16- and even several 8-bit processors.

µC/OS-III Goals

The main goal of µC/OS-III is to provide a best-in-class real-time kernel that literally shaves months of development time from an embedded-product schedule. Using a commercial real-time kernel such as µC/OS-III provides a solid foundation and framework to the design engineer dealing with the growing complexity of embedded designs.

Another goal for µC/OS-III, and therefore this documentation, is to explain inner workings of a commercial-grade kernel. This understanding will assist the reader in making logical design decisions and informed tradeoffs between hardware and software that make sense.

Copyright 2015 Micrium Inc.

Intended Audience

This documentation is written for embedded systems programmers, consultants, hobbyists and students interested in understanding the inner workings of a real-time kernel. µC/OS-III is not just a great learning platform, but also a commercial-grade software package ready to be part of a range of products.

To get the most from this documentation, it is assumed that the reader has a good working knowledge of microprocessors, microcontrollers, and/or Digital Signal Processors (DSPs). That knowledge should extend to CPU instructions, interrupts, I/O devices, RAM and ROM or Flash, memory addresses, and stack pointers.

It is also expected that the reader will have a good working knowledge of the C programming language and assembly language.

The µC/OS Story

The µC/OS story started when, in 1989, I joined Dynalco Controls in Fort Lauderdale, Florida, and began working on the design of a new microprocessor-based ignition control system for a large industrial reciprocating engine.

Given that I had experience using a kernel, I was convinced that an operating system would substantially benefit this project and other planned projects at Dynalco. Time to market was of paramount importance for this ignition control system, and I knew that a kernel would help meet scheduling requirement. I also knew that new features would be added to this product in the future, and a preemptive operating system would allow for such updates without negatively impacting system responsiveness.

The kernel that I initially considered was one that had served me well in the past. However, it carried a hefty price tag, and my budget was somewhat meager. The alternative was a kernel that I had not used before, but it was five-times cheaper than my original choice. Ultimately, I decided that the financial benefits of using the unfamiliar operating system outweighed the potential advantages that its higher-priced counterpart could offer.

I quickly realized, however, that I would pay for the seemingly cheaper operating system with my time. During the two months after receiving the kernel, I was in constant contact with technical support, trying fruitlessly to determine why even the simplest applications would not

Copyright 2015 Micrium Inc.

run. The operating system, said to be written in C, required me to initialize all of the kernel's internal variables in assembly language, a task laden with problems. I eventually discovered that I was one of the first customers to purchase this operating system and was essentially an unknowing beta tester.

Frustrated with the software's many shortcomings, I turned to the relatively expensive operating system that I originally rejected. It seems that if a project is late, money is no object. Within two days, I was running simple applications that the cheap operating system seemed unable to support. My kernel-related problems seemed to be over.

Before long, however, I would find myself at another impasse. My second set of problems began when one of my engineers reported that the new operating system seemed to contain a bug. I promptly relayed the engineer's findings to the software vendor, assuming that the company would be interested. Instead of receiving assurance that the bug would be fixed, I was notified that the 90-day warranty expired. Unless I purchased a maintenance contract, the bug would not be eliminated, which was absurd to me. The software provider thought otherwise, and I forked over the maintenance fee.

Incredibly, the vendor took six months to actually remove the bug. All told, I completed my ignition system, incorporating the second operating system, a year after receiving the software. Clearly, I needed a better solution.

Twice disappointed, I began to develop my own kernel. In my naive opinion, all a kernel really did was to save and restore CPU registers; writing one should not be especially challenging.

The project kept me busy at night and on weekends, and proved to be much more difficult than anticipated. Approximately a year after I starting the project, my first operating system was complete.

With a new kernel in hand, there was finally a handy means of developing multitasking applications. The operating system, consisted of little more than a single C file and allowed up to 64 tasks to be created in a single application. Each task was required to be have a unique priority. The highest priority task that was ready to run when the operating system's scheduler was invoked was given control of the CPU. µC/OS was preemptive, so scheduling could occur at practically any time.

Efficient task scheduling was actually one of many services offered by µC/OS. The operating system also facilitated inter-task communication (via message queues and mailboxes) and task

Copyright 2015 Micrium Inc.

synchronization (through semaphores). All elements of µC/OS were designed to be both highly dependable and easy to use.

Presumably, most kernel developers have similar goals in mind when they write new software. I was especially well equipped to meet these goals, in part because of my punctilious coding style. Throughout my career, I focused on consistency and documentation. I began using formal coding standards in 1984, and the consistency of the µC/OS code is a testimony to this process.

µC/OS was designed according to the stringent standards that I created and promulgated at Dynalco. The operating system's source code featured liberal spacing, carefully worded comments, and consistent naming. Offering further evidence of the prudent coding techniques, the kernel was also highly portable. Although µC/OS, like its kernel peers, featured a small number of processor-specific functions, these routines were clearly separated from other portions of the operating system. Engineers could easily adapt µC/OS to new CPU architectures.

Unfortunately, I was the only one to know about the virtues of µC/OS. Eager to describe my new software to others, I wrote an in-depth paper explaining the inner workings of µC/OS. There was plenty to say, and my final paper was approximately 70 pages in length.

I offered my paper to C User's Journal, and they rejected it on the grounds that it was too long and that its subject matter wasn't fresh. The magazine had already published several kernel articles, and this was just one more. Convinced that my article was unique, I offered it to Embedded Systems Programming. The editor of this periodical likewise expressed misgivings, but I convinced him that µC/OS was attention-worthy. I explained that the operating system was comparable in quality to products that major embedded software companies offered (and better than at least two). I also explained that the source code for µC/OS could actually be placed on the publication's bulletin board service (BBS).

Embedded Systems Programming published a trimmed-down version of the paper as a two-part series. Both issues generated strong responses. Engineers were grateful that the inner workings of a high-quality kernel were revealed, and they downloaded the µC/OS source code in droves. kernel vendors, on the other hand, were less than thrilled with the article. In fact, the vendor of the low cost kernel was especially upset claiming that I had copied his work. Imagine that I would base µC/OS on software that didn't work!

There would soon be even more reason for the kernel vendors to be upset. Shortly after my

Copyright 2015 Micrium Inc.

article appeared in Embedded Systems Programming, R & D Publications, publisher of C User's Journal contacted me, and they were interested in printing an entire µC/OS book.

Originally, the plan for the book simply involved printing all of the material that I had originally submitted to C User's Journal. Had I taken that route, the resulting book would have been approximately 80 pages or so in length. To make the most of this opportunity, I prepared a comprehensive text. With the consent of R & D, I spent the next several months writing. In late 1992, my first book, aptly titled µC/OS, The Real-Time Kernel, was released. The book had 250 pages, and was available in paperback form.

Although initial sales of the book were somewhat disappointing, R & D advertised µC/OS, The Real-Time Kernel each month in C User's Journal. At the same time, I was beginning to gain attention as a kernel expert. In the spring of 1993, I was invited to speak at the Embedded Systems Conference (ESC) in Atlanta, Georgia, where I described operating system fundamentals to a highly receptive audience of more than 70 embedded enthusiasts. Within a few years, I was an ESC fixture, delivering my kernel lectures to hundreds of engineers at each conference.

While my popularity as a speaker rose, interest in my book also picked up steam. After its slow start, µC/OS, The Real-Time Kernel, went on to sell more than 15,000 copies.

Thanks to the success of my book, the number of engineers using µC/OS increased substantially throughout the 1990s. Developers easily adapted the operating system to new hardware platforms, and were designing a myriad of µC/OS-based applications. Although several µC/OS users simply tinkered with the operating system in their spare time, many engineers used the software commercially in complex and demanding projects. Comments and suggestions from µC/OS users helped me to continue to refine and evolve the operating system.

For several years, only minor changes were made to µC/OS. However, when R & D asked me to write a second edition, I decided that a substantial update of both the operating system and the book was warranted. The updated operating system became µC/OS-II.

A quick glance at the µC/OS-II files revealed that this operating system was different from µC/OS. Whereas all of the processor-independent code incorporated by µC/OS was contained in a single C file, µC/OS-II spanned multiple files, each corresponding to one of the operating system's services. µC/OS-II also offered many features that its predecessor lacked, including stack-checking capabilities, hook functions, and a safe means to dynamically allocate memory.

Copyright 2015 Micrium Inc.

To fully describe all of the new operating system's features, I nearly doubled the size of the book. Just as the latest version of the software received a new name, the new edition became MicroC/OS-II, The Real-Time Kernel. ("Micro" was used in place of "µ" because titles incorporating Greek letters posed problems for many book retailers.) Unlike my first text, the new book would be a hardcover.

MicroC/OS-II, The Real-Time Kernel was released in 1998. This new text was accompanied by the source code that it described and I would again have thousands of developers testing the kernel and providing valuable feedback.

Among the thousands of readers of my books using the software, there were many kernel rookies. For them, the book provided thorough and accessible coverage of operating system fundamentals. Many university professors recognized the book's appeal to new kernel users and started designing entire courses around µC/OS-II. Soon college graduates whose kernel training focused on the operating system made their way into the workforce, where they continued to use µC/OS-II.

While students gravitated to µC/OS-II because of my book and readily available source code, a substantial number of engineers using µC/OS-II commercially selected the software for its reliability. Definitive proof of the operating system's reliability was provided in July 2000, when DO-178B Level A certification was conferred on an avionics product incorporating µC/OS-II. This certification, recognized by the Federal Avionics Administration (FAA), is awarded to software deemed safe enough to be used in aircraft. To this day, there are few operating systems that have successfully completed the rigorous testing that certified software must undergo.

DO-178B certification is only one of µC/OS-II's credentials. Additional certifications include Food and Drug Administration (FDA) pre-market notification (510(k)), pre-market approval (PMA) for medical devices, and IEC-61508 for industrial controls. Compliance with such standards is critical within industry segments; however, the certifications also have value for engineers in other industries as they evidence reliability, documentation, and time-to-market advantages beneficial to any design.

As the decade came to a close, I still worked full time at Dynalco, and experienced difficulty keeping up with the demand for the operating system. I felt obligated to respond to each µC/OS-II user that contacted me, and the flow of messages into my inbox was unrelenting. Since I could no longer treat the operating system as a side project, I made the decision to found my own software company. In September 1999, Micriµm, officially came into being.

Copyright 2015 Micrium Inc.

Micriμm comes from the word ‘Micro’ (for microprocessors or microcontrollers) and ‘ium’ (which means the Universe of) and thus, Micriμm means the Universe of Microprocessors (as seen through the eyes of software).

In the months before incorporating Micriμm, I began working on a second edition of the µC/OS-II book, which made its debut in November 1999 and was accompanied by a new version of the kernel. Two major features to the operating system were added: event flags and mutual exclusion semaphores. These new features, fully described in the book, were heartily welcomed by µC/OS-II users. The book itself was similarly embraced; the second edition of MicroC/OS-II, The Real-Time Kernel quickly became common sight on the bookshelves of embedded software developers. In fact, the MicroC/OS-II book is the most popular embedded systems book ever sold.

Micriμm expanded. Engineers were hired to adapt µC/OS-II to new hardware platforms and develop a bevy of example projects and application notes. A long-time friend of mine, Christian Legare joined Micriμm as Vice President in 2002, and his substantial corporate and technical expertise further accelerated the company’s rapid growth. Since Christian joined Micriμm, the company expanded from a one-product company to one with a portfolio of 15 products.

Meanwhile, new features were added to satisfy the ever-evolving needs of µC/OS-II users, including a variety of new API functions to the operating system and expanding the maximum number of tasks supported by the kernel from 64 to 255.

As Micriμm’s president, I remain dedicated to writing world-class kernel code, most recently µC/OS-III. The product of countless hours of meticulous programming and testing, this robust operating system has its roots in µC/OS-II, yet is an entirely new kernel. Addressing input received from customers and all of the lessons learned along the way, several additional important µC/OS-III features were included (see Introduction).

I am highly circumspect of fads and unproven technology as I write new software. Although I like to keep abreast of the latest developments in the high-tech world, the focus is on solving engineers’ problems and providing a solid and complete infrastructure, rather than on how to prematurely exploit emerging trends.

This philosophy has yielded considerable success. Micriμm, now in its 14th year, is a highly respected embedded software provider. Industry surveys consistently show the operating systems to be among the most popular in the embedded space. My goal has always been, and

Copyright 2015 Micrium Inc.

continues to be to provide effective solutions for the same types of problems that I confronted at Dynalco, and that millions of embedded systems developers continue to face today.

Acknowledgements

First and foremost, I'd like to thank my loving and caring wife Manon for her unconditional support, encouragement, understanding and patience. This project was again a huge undertaking, and I could not have done it without her.

I would also like to thank many fine people at Micriµm who have tested the code and reviewed the documentation. In alphabetic order:

- Brian Nagel
- Eric Shufro
- Hong Soong
- Freddy Torres

A special thanks to Frank Voorburg from Feaser and to Ian Hall and Robert Mongrain from Renesas for feedback and corrections to the text to Michael Barr for sharing his real life RTOS experiences, and to Carolyn Mathas for the incredible job of editing this huge project.

A very special thanks to my long-time friend, colleague and partner, Christian Legare, who has provided his advice and support throughout this project and on a day-to-day basis at Micrium. Thank you also to the dozens of people who provided feedback about the µC/OS-III code, as well as reviewers.

Finally, I listen to music when I write software, and artist Gino Vannelli's awesome music has provided a creative environment for me for over three decades. I would be remiss if I did not acknowledge his contribution here as well.

Copyright 2015 Micrium Inc.

Introduction

Real-time systems are systems whereby the correctness of the computed values and their timeliness are at the forefront. There are two types of real-time systems, hard and soft real time.

What differentiates hard and soft real-time systems is their tolerance to missing deadlines and the consequences associated with those misses. Correctly computed values after a deadline has passed are often useless.

For hard real-time systems, missing deadlines is not an option. In fact, in many cases, missing a deadline often results in catastrophe, which may involve human lives. For soft real-time systems, however, missing deadlines is generally not as critical.

Real-time applications cover a wide range, but many real-time systems are embedded. An embedded system is a computer built into a system and not acknowledged by the user as being a computer. Embedded systems are also typically dedicated systems. In other words, systems that are designed to perform a dedicated function. The following list shows just a few examples of embedded systems:

Aerospace • Flight management systems • Jet engine controls • Weapons systems **Audio** • MP3 players • Amplifiers and tuners **Automotive** • Antilock braking systems • Climate control • Engine controls • Navigation systems (GPS)	**Communications** • Routers • Switches • Cell phones **Computger peripherals** • Printers • Scanners **Domestic** • Air conditioning units • Thermostats • White goods	**Office automation** • FAX machines / copiers **Process control** • Chemical plants • Factory automation • Food processing **Robots** **Video** • Broadcasting equipment • HD Televisions **And many more**

Real-time systems are typically more complicated to design, debug, and deploy than

Copyright 2015 Micrium Inc.

non-real-time systems.

Copyright 2015 Micrium Inc.

Foreground-Background Systems

Small systems of low complexity are typically designed as foreground/background systems or super-loops. An application consists of an infinite loop that calls modules (i.e., tasks) to perform the desired operations (background). Interrupt Service Routines (ISRs) handle asynchronous events (foreground). Foreground is also called interrupt level; background is called task level.

Critical operations that should be performed at the task level must unfortunately be handled by the ISRs to ensure that they are dealt with in a timely fashion. This causes ISRs to take longer than they should. Also, information for a background module that an ISR makes available is not processed until the background routine gets its turn to execute, which is called the task-level response. The worst-case task-level response time depends on how long a background loop takes to execute and, since the execution time of typical code is not constant, the time for successive passes through a portion of the loop is nondeterministic. Furthermore, if a code change is made, the timing of the loop is affected.

Most high-volume and low-cost microcontroller-based applications (e.g., microwave ovens, telephones, toys, etc.) are designed as foreground/background systems.

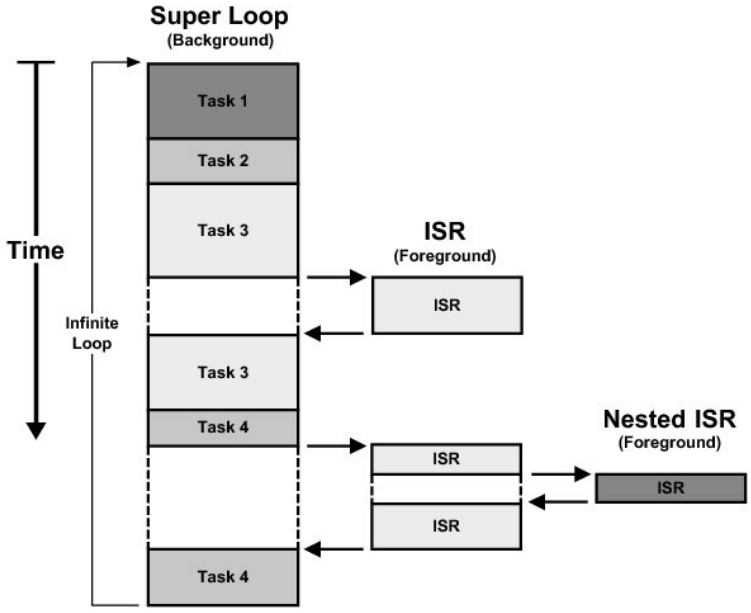


Figure - Foreground/Background (SuperLoops) systems

Real-Time Kernels

A real-time kernel is software that manages the time and resources of a microprocessor, microcontroller or Digital Signal Processor (DSP).

The design process of a real-time application involves splitting the work into tasks, each responsible for a portion of the job. A task (also called a thread) is a simple program that thinks it has the Central Processing Unit (CPU) completely to itself. On a single CPU, only one task executes at any given time. A task is also typically implemented as an infinite loop.

The kernel is responsible for the management of tasks. This is called multitasking. Multitasking is the process of scheduling and switching the CPU between several tasks. The CPU switches its attention between several sequential tasks. Multitasking provides the illusion of having multiple CPUs and maximizes the use of the CPU. Multitasking also helps in the creation of modular applications. One of the most important aspects of multitasking is that it allows the application programmer to manage the complexity inherent in real-time applications. Application programs are easier to design and maintain when multitasking is used.

µC/OS-III is a preemptive kernel, which means that µC/OS-III always runs the most important task that is ready-to-run as shown in Figure 1-2.

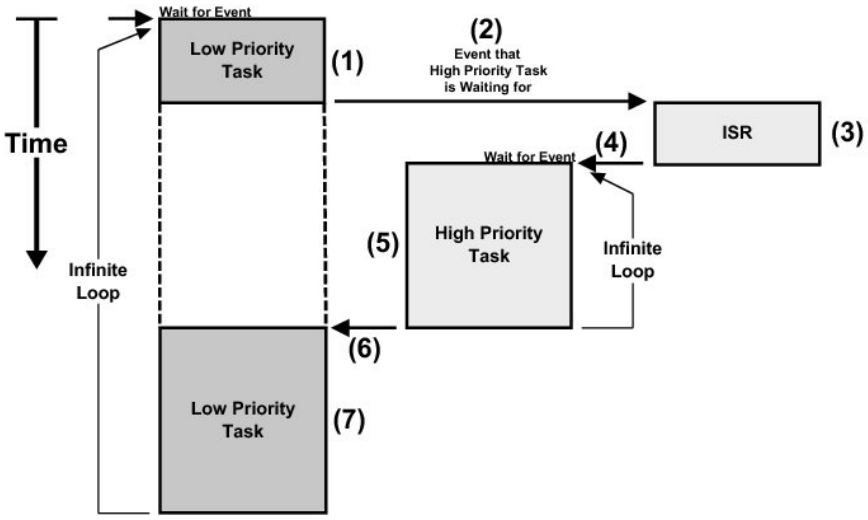


Figure - µC/OS-III is a preemptive kernel

(1) A low-priority task is executing.

(2) An interrupt occurs, and the CPU vectors to the ISR responsible for servicing the interrupting device.

(3) The ISR services the interrupt device, but actually does very little work. The ISR will typically signal or send a message to a higher-priority task that will be responsible for most of the processing of the interrupting device. For example, if the interrupt comes from an Ethernet controller, the ISR simply signals a task, which will process the received packet.

(4) When the ISR finishes, µC/OS-III notices that a more important task has been made ready-to-run by the ISR and will not return to the interrupted task, but instead context switch to the more important task.

(5) The higher-priority task executes and performs the necessary processing in response to the interrupt device.

(6) When the higher-priority task completes its work, it loops back to the beginning of the task code and makes a µC/OS-III function call to wait for the next interrupt from the device.

(7) The low-priority task resumes exactly at the point where it was interrupted, not knowing what happened.

Kernels such as µC/OS-III are also responsible for managing communication between tasks, and managing system resources (memory and I/O devices).

A kernel adds overhead to a system because the services provided by the kernel require time to execute. The amount of overhead depends on how often these services are invoked. In a well-designed application, a kernel uses between 2% and 4% of a CPU’s time. And, since µC/OS-III is software that is added to an application, it requires extra ROM (code space) and RAM (data space).

Low-end single-chip microcontrollers are generally not able to run a real-time kernel such as µC/OS-III since they have access to very little RAM. µC/OS-III requires between 1 Kbyte and

Copyright 2015 Micrium Inc.

4 Kbytes of RAM, plus each task requires its own stack space. It is possible for µC/OS-III to work on processors having as little as 4 Kbytes of RAM.

Finally, µC/OS-III allows for better use of the CPU by providing approximately 70 indispensable services. After designing a system using a real-time kernel such as µC/OS-III, you will not return to designing a foreground/background system.

Copyright 2015 Micrium Inc.

RTOS Real-Time Operating System

A Real Time Operating System generally contains a real-time kernel and other higher-level services such as file management, protocol stacks, a Graphical User Interface (GUI), and other components. Most additional services revolve around I/O devices.

Micriμm offers a complete suite of RTOS components including: µC/FS (an Embedded File System), µC/TCP-IP (a TCP/IP stack), µC/GUI (a Graphical User Interface), two USB stacks, µC/USB-Device and µC/USB-Host, and more. Most of these components are designed to work standalone, users can pick and choose only the components required for the application. Contact Micriμm (www.micrium.com) for additional details and pricing.

Copyright 2015 Micrium Inc.

uC-OS-III

µC/OS-III is a scalable, ROMable, preemptive real-time kernel that manages an unlimited number of tasks. µC/OS-III is a third-generation kernel, offering all of the services expected from a modern real-time kernel including resource management, synchronization, inter-task communication, and more. However, µC/OS-III also offers many unique features not found in other real-time kernels, such as the ability to perform performance measurements at run time, directly signal or send messages to tasks, and pending (i.e., waiting) on such multiple kernel objects as semaphores and message queues.

Here is a list of features provided by µC/OS-III:

Source Code: µC/OS-III is provided in ANSI-C source form. The source code for µC/OS-III is arguably the cleanest and most consistent kernel code available. Clean source is part of the corporate culture at Micriµm. Although many commercial kernel vendors provide source code for their products, unless the code follows strict coding standards and is accompanied by complete documentation with examples to show how the code works, these products may be cumbersome and difficult to harness. With this book, you will gain a deep understanding of the inner workings of µC/OS-III, which will protect your investment.

Intuitive Application Programming Interface (API): µC/OS-III is highly intuitive. Once familiar with the consistent coding conventions used, it is simple to predict the functions to call for the services required, and even predict which arguments are needed. For example, a pointer to an object is always the first argument, and a pointer to an error code is always the last one.

Preemptive multitasking: µC/OS-III is a preemptive multi-tasking kernel and therefore, µC/OS-III always runs the most important ready-to-run task.

Round robin scheduling of tasks at equal priority: µC/OS-III allows multiple tasks to run at the same priority level. When multiple tasks at the same priority are ready-to-run, and that priority level is the most important level, µC/OS-III runs each task for a user-specified time called a time quanta. Each task can define its own time quanta, and a task can also give up the CPU to another task at the same priority if it does not require the full time quanta.

Low interrupt disable time: µC/OS-III has a number of internal data structures and variables that it needs to access atomically. To ensure this, µC/OS-III is able to protect these critical

Copyright 2015 Micrium Inc.

regions by locking the scheduler instead of disabling interrupts. Interrupts are therefore disabled for very little time. This ensures that µC/OS-III is able to respond to some of the fastest interrupt sources.

Deterministic: Interrupt response with µC/OS-III is deterministic. Also, execution times of most services provided by µC/OS-III are deterministic.

Scalable: The footprint (both code and data) can be adjusted based on the requirements of the application. Adding and removing features (i.e., services) is performed at compile time through more than 50 `#defines` (see `os_cfg.h`). µC/OS-III also performs a number of run-time checks on arguments passed to µC/OS-III services. Specifically, µC/OS-III verifies that the user is not passing `NULL` pointers, not calling task level services from ISRs, that arguments are within allowable range, and options specified are valid, etc.. These checks can be disabled (at compile time) to further reduce the code footprint and improve performance. The fact that µC/OS-III is scalable allows it to be used in a wide range of applications and projects.

Portable: µC/OS-III can be ported to a large number of CPU architectures. Most µC/OS-II ports are easily converted to work on µC/OS-III with minimal changes in just a matter of minutes and therefore benefit from more than 45 CPU architectures already supported by µC/OS-II.

ROMable: µC/OS-III was designed especially for embedded systems and can be ROMed along with the application code.

Run-time configurable: µC/OS-III allows the user to configure the kernel at run time. Specifically, all kernel objects such as tasks, stacks, semaphores, event-flag groups, message queues, number of messages, mutual exclusion semaphores, memory partitions and timers, are allocated by the user at run time. This prevents over-allocating resources at compile time.

Unlimited number of tasks: µC/OS-III supports an unlimited number of tasks. From a practical standpoint, however, the number of tasks is actually limited by the amount of memory (both code and data space) that the processor has access to. Each task requires its own stack space and, µC/OS-III provides features to allow stack growth of the tasks to be monitored at run-time.

µC/OS-III does not impose any limitations on the size of each task, except that there be a minimum size based on the CPU used.

Copyright 2015 Micrium Inc.

Unlimited number of priorities: µC/OS-III supports an unlimited number of priority levels. However, configuring µC/OS-III for between 32 and 256 different priority levels is more than adequate for most applications.

Unlimited number of kernel objects: µC/OS-III allows for any number of tasks, semaphores, mutual exclusion semaphores, event flags, message queues, timers, and memory partitions. The user allocates all kernel objects at run-time.

Services: µC/OS-III provides all the services expected from a high-end real-time kernel, such as task management, time management, semaphores, event flags, mutexes, message queues, software timers, fixed-size memory pools, etc.

Mutual Exclusion Semaphores (Mutexes): Mutexes are provided for resource management. Mutexes are special types of semaphores that have built-in priority inheritance, which eliminate unbounded priority inversions. Accesses to a mutex can be nested and therefore, a task can acquire the same mutex up to 250 times. Of course, the mutex owner needs to release the mutex an equal number of times.

Nested task suspension: µC/OS-III allows a task to suspend itself or another task. Suspending a task means that the task will not be allowed to execute until the task is resumed by another task. Suspension can be nested up to 250 levels deep. In other words, a task can suspend another task up to 250 times. Of course, the task must be resumed an equal number of times for it to become eligible to run on the CPU.

Software timers: You can define any number of “one-shot” and/or “periodic” timers. Timers are countdown counters that perform a user-definable action upon counting down to 0. Each timer can have its own action and, if a timer is periodic, the timer is automatically reloaded and the action is executed every time the countdown reaches zero.

Pend on multiple objects: µC/OS-III allows an application to wait (i.e., pend) on multiple events at the same time. Specifically, a task can wait on multiple semaphores and/or message queues to be posted. The waiting task wakes up as soon as one of the events occurs.

Task Signals: µC/OS-III allows an ISR or task to directly signal a task. This avoids having to create an intermediate kernel object such as a semaphore or event flag just to signal a task, and results in better performance.

Task Messages: µC/OS-III allows an ISR or a task to send messages directly to a task. This

Copyright 2015 Micrium Inc.

avoids having to create and use a message queue, and also results in better performance.

Task registers: Each task can have a user-definable number of “task registers.” Task registers are different than CPU registers. Task registers can be used to hold “errno” type variable, IDs, interrupt disable time measurement on a per-task basis, and more.

Error checking: µC/OS-III verifies that `NULL` pointers are not passed, that the user is not calling task-level services from ISRs, that arguments are within allowable range, that options specified are valid, that a pointer to the proper object is passed as part of the arguments to services that manipulate the desired object, and more. Each µC/OS-III API function returns an error code concerning the outcome of the function call.

Built-in performance measurements: µC/OS-III has built-in features to measure the execution time of each task, stack usage of each task, number of times a task executes, CPU usage, ISR-to-task and task-to-task response time, peak number of entries in certain lists, interrupt disable and scheduler lock time on a per-task basis, and more.

Can easily be optimized: µC/OS-III was designed so that it could easily be optimized based on the CPU architecture. Most data types used in µC/OS-III can be changed to make better use of the CPU’s natural word size. Also, the priority resolution algorithm can easily be written in assembly language to benefit from special instructions such as bit set and clear, as well as count-leading-zeros (CLZ), or find-first-one (FF1) instructions.

Deadlock prevention: All of the µC/OS-III “pend” services include timeouts, which help avoid deadlocks.

Tick handling at task level: The clock tick manager in µC/OS-III is accomplished by a task that receives a trigger from an ISR. Handling delays and timeouts by a task greatly reduces interrupt latency. Also, µC/OS-III uses a hashed delta list mechanism, which further reduces the amount of overhead in processing delays and timeouts of tasks.

User definable hooks: µC/OS-III allows the port and application programmer to define “hook” functions, which are called by µC/OS-III. A hook is simply a defined function that allows the user to extend the functionality of µC/OS-III. One such hook is called during a context switch, another when a task is created, yet another when a task is deleted, etc.

Timestamps: For time measurements, µC/OS-III requires that a 16-bit or 32-bit free running counter be made available. This counter can be read at run time to make time measurements of

Copyright 2015 Micrium Inc.

certain events. For example, when an ISR posts a message to a task, the timestamp counter is automatically read and saved as part of the message posted. When the recipient receives the message, the timestamp is provided to the recipient, and by reading the current timestamp, the time it took for the message to be received can be determined.

Built-in support for Kernel Awareness debuggers: This feature allows kernel awareness debuggers to examine and display µC/OS-III variables and data structures in a user-friendly way. The kernel awareness support in µC/OS-III can be used by µC/Probe to display this information at run-time.

Object names: Each µC/OS-III kernel object can have a name associated with it. This makes it easy to recognize what the object is assigned to. You can thus assign an ASCII name to a task, a semaphore, a mutex, an event flag group, a message queue, a memory partition, and a timer. The object name can have any length, but must be NUL terminated.

Copyright 2015 Micrium Inc.

uC-OS uC-OS-II and uC-OS-III Features Comparison

The table below shows the evolution of µC/OS over the years, comparing the features available in each version.

Feature	µC/OS	µC/OS-II	>µC/OS-III
Year introduced	1992	1998	2009
Book	Yes	Yes	Yes
Source code available	Yes	Yes	Yes
Preemptive Multitasking	Yes	Yes	Yes
Maximum number of tasks	64	255	Unlimited
Number of tasks at each priority level	1	1	Unlimited
Round Robin Scheduling	No	No	Yes
Semaphores	Yes	Yes	Yes
Mutual Exclusion Semaphores	No	Yes	Yes (Nestable)
Event Flags	No	Yes	Yes
Message Mailboxes	Yes	Yes	No (not needed)
Message Queues	Yes	Yes	Yes
Fixed Sized Memory Management	No	Yes	Yes
Signal a task without requiring a semaphore	No	No	Yes
Option to Post without scheduling	No	No	Yes
Send messages to a task without requiring a message queue	No	No	Yes
Software Timers	No	Yes	Yes
Task suspend/resume	No	Yes	Yes (Nestable)
Deadlock prevention	Yes	Yes	Yes
Scalable	Yes	Yes	Yes
Code Footprint	3K to 8K	6K to 26K	6K to 24K
Data Footprint	1K+	1K+	1K+
ROMable	Yes	Yes	Yes
Run-time configurable	No	No	Yes
Compile-time configurable	Yes	Yes	Yes
ASCII names for each kernel object	No	Yes	Yes

Copyright 2015 Micrium Inc.

Pend on multiple objects	No	Yes	Yes
Task registers	No	Yes	Yes
Built-in performance measurements	No	Limited	Extensive
User definable hook functions	No	Yes	Yes
Time stamps on posts	No	No	Yes
Built-in Kernel Awareness support	No	Yes	Yes
Optimizable Scheduler in assembly language	No	No	Yes
Catch a task that returns	No	No	Yes
Tick handling at task level	No	No	Yes
Source code available	Yes	Yes	Yes
Number of services	~20	~90	~70
MISRA-C:1998	No	Yes (except 10 rules)	N/A
MISRA-C:2004	No	No	Yes (except 7 rules)
DO178B Level A and EUROCAE ED-12B	No	Yes	Yes
Medical FDA pre-market notification (510(k)) and pre-market approval (PMA)	No	Yes	Yes
SIL3/SIL4 IEC for transportation and nuclear systems	No	Yes	Yes
IEC-61508	No	Yes	Yes

Table - 1-1 µC/OS, µC/OS-II and µC/OS-III Features Comparison Chart

uC-Probe

µC/Probe is a Microsoft Windows™ based application that enables the user to visualize variables in a target at run time. Specifically, you can display or change the value of any variable in a system while the target is running. These variables can be displayed using such graphical elements as gauges, meters, bar graphs, virtual LEDs, numeric indicators, and many more. Sliders, switches, and buttons can be used to change variables. This is accomplished without the user having to write a single line of code!

µC/Probe interfaces to any target (8-, 16-, 32-, 64-bit, or even DSPs) through one of the many interfaces supported (J-Tag, RS-232C, USB, Ethernet, etc.). µC/Probe displays or changes any variable (as long as they are global) in the application, including µC/OS-III's internal variables.

µC/Probe works with any compiler/assembler/linker able to generate an ELF/DWARF or IEEE695 file. This is the exact same file that the user will download to the evaluation board or a final target. From this file, µC/Probe is able to extract symbolic information about variables, and determine where variables are stored in RAM or ROM.

µC/Probe also allows users to log the data displayed into a file for analysis of the collected data at a later time. µC/Probe also provides µC/OS-III kernel awareness as a built-in feature.

The trial version that accompanies the book is limited to the display or change of up to eight (8) variables.

µC/Probe is a tool that serious embedded software engineers should have in their toolbox. The full version of µC/Probe is available from Micriµm, see www.micrium.com for more details.

Copyright 2015 Micrium Inc.

Documentation Conventions

There are a number of conventions in this documentation.

You will notice that when a specific element in a figure is referenced, the element has a number next to it in parenthesis. A description of this element follows the figure.

Code quality is something I've been avidly promoting throughout my whole career. At Micrim, we pride ourselves in having the cleanest code in the industry. Examples of this are seen in this book. I created and published a coding standard in 1992 that was published in the original C/OS book. This standard has evolved over the years, but the spirit of the standard has been maintained throughout. The Micrim coding standard is available for download from the Micrim website, www.micrium.com.

One of the conventions used is that all functions, variables, macros and `#define` constants are prefixed by "OS" (which stands for Operating System) followed by the acronym of the module (e.g., `Sem`), and then the operation performed by the function. For example `OSSemPost()` indicates that the function belongs to the OS (C/OS-III), that it is part of the Semaphore services, and specifically that the function performs a `Post` (i.e., signal) operation. This allows all related functions to be grouped together in the reference manual, and makes those services intuitive to use.

You should notice that signaling or sending a message to a task is called posting, and waiting for a signal or a message is called pending. In other words, an ISR or a task signals or sends a message to another task by using `OS???Post()`, where `???` is the type of service: `Sem`, `TaskSem`, `Flag`, `Mutex`, `Q`, and `TaskQ`. Similarly, a task can wait for a signal or a message by calling `OS???Pend()`.

Copyright 2015 Micrium Inc.

Directories and Files

µC/OS-III is fairly easy to use once it is understood exactly which source files are needed to make up a µC/OS-III-based application. This chapter will discuss the modules available for µC/OS-III and how everything fits together.

The figure below shows the µC/OS-III architecture and its relationship with hardware. Of course, in addition to the timer and interrupt controller, hardware would most likely contain such other devices as Universal Asynchronous Receiver Transmitters (UARTs), Analog to Digital Converters (ADCs), Ethernet controller(s) and more.

This chapter assumes development on a Windows®-based platform and makes references to typical Windows-type directory structures (also called Folder). However, since µC/OS-III is provided in source form, it can also be used on Unix, Linux or other development platforms.

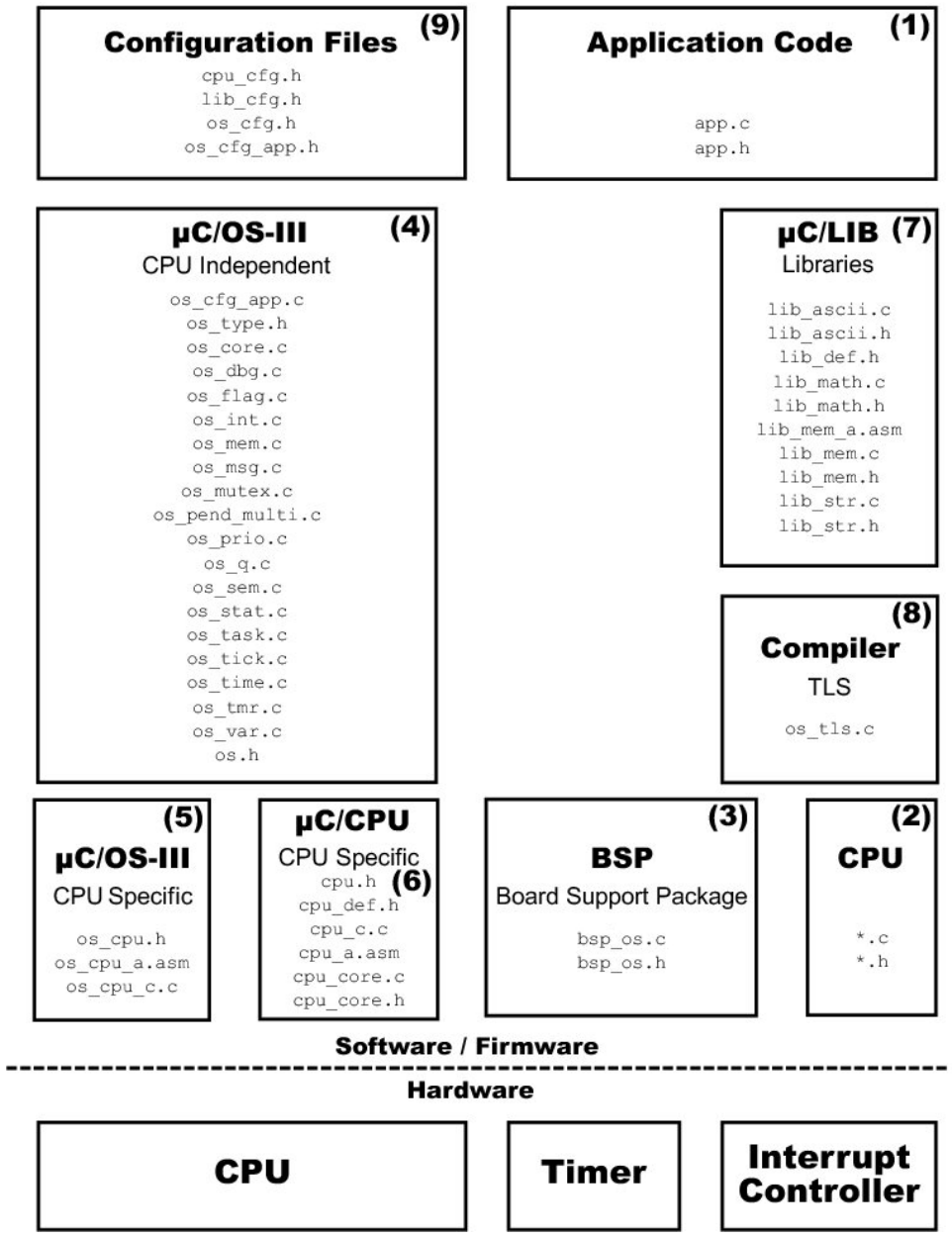


Figure - µC/OS-III Architecture

(1) The application code consists of project or product files. For convenience, these are simply called `app.c` and `app.h`, however an application can contain any number of files that do not have to be called app.*. The application code is typically where one would find the `main()`.

(2) Semiconductor manufacturers often provide library functions in source form for accessing the peripherals on their CPU or MCU. These libraries are quite useful and often save valuable time. Since there is no naming convention for these files, `*.c` and `*.h` are assumed.

(3) The Board Support Package (BSP) code needed by µC/OS-III is typically quite simple and generally, µC/OS-III only requires that you initialize a periodic interrupt source which is used for time delays and timeouts. This functionality can should be placed in a file called os_bsp.c along with its corresponding header file, `bsp_os.h`.

(4) This is the µC/OS-III processor-independent code. This code is written in highly portable ANSI C.

(5) This is the µC/OS-III code that is adapted to a specific CPU architecture and is called a port. µC/OS-III has its roots in µC/OS-II and benefits from being able to use most of the 45 or so ports available for µC/OS-II. µC/OS-II ports, however, will require small changes to work with µC/OS-III.

(6) At Micriμm, we encapsulate CPU functionality. These files define functions to disable and enable interrupts, `CPU_???` data types to be independent of the CPU and compiler used, and many more functions.

(7) µC/LIB is of a series of source files that provide common functions such as memory copy, string, and ASCII-related functions. Some are occasionally used to replace `stdlib` functions provided by the compiler. The files are provided to ensure that they are fully portable from application to application and especially, from compiler to compiler. µC/OS-III does not use these files, but µC/CPU does.

(8) Some compilers provide extensions to make library functions ‘thread safe’. The adaptation of those functions is performed in the `os_tls.c` file which is found under the `uCOS-III\TLS\<tool>` folder. `<tool>` is the name of the tool manufacturer or, the tool name.

(9) Configuration files are used to define µC/OS-III features (`os_cfg.h`) to include in the application, specify the size of certain variables and data structures expected by

Copyright 2015 Micrium Inc.

µC/OS-III (`os_cfg_app.h`), such as idle task stack size, tick rate, size of the message pool, configure the µC/CPU features available to the application programmer (`cpu_cfg.h`) and also configure µC/LIB options (`lib_cfg.h`).

Copyright 2015 Micrium Inc.

Application Code

When Micriμm provides example projects, they are placed in a directory structure shown below. Of course, a directory structure that suits a particular project/product can also be used.

```
\Micrium
    \Software
        \EvalBoards
            \<manufacturer>
                \<board_name>
                    \<compiler>
                        \<project name>
                            \*.*
```

`\Micrium`

This is where we place all software components and projects provided by Micriμm. This directory generally starts from the root directory of the computer.

`\Software`

This sub-directory contains all software components and projects.

`\EvalBoards`

This sub-directory contains all projects related to evaluation boards supported by Micriμm.

`\<manufacturer>`

This is the name of the manufacturer of the evaluation board. The “`<`” and “`>`” are not part of the actual name.

`\<board name>`

This is the name of the evaluation board. A board from Micriμm will typically be called `uC?Eval?xxxx` where “`xxxx`” represents the CPU or MCU used on the board. The “`<`” and “`>`” are not part of the actual name.

`\<compiler>`

Copyright 2015 Micrium Inc.

This is the name of the compiler or compiler manufacturer used to build the code for the evaluation board. The “<” and “>” are not part of the actual name.

`\<project name>`

The name of the project that will be demonstrated. For example, a simple µC/OS-III project might have a project name of “`OS-Ex1`”. The “`-Ex1`” represents a project containing only µC/OS-III.

`\*.*`

These are the project source files. Main files can optionally be called `app*.*`. This directory also contains configuration files `os_cfg.h`, `os_cfg_app.h` and other required source files.

Copyright 2015 Micrium Inc.

uC-OS-III Board Support Package BSP

The Board Support Package (BSP) is generally found with the evaluation or target board as it is specific to that board. In fact, when well written, the BSP should be used for multiple projects.

```
\Micrium
    \Software
        \EvalBoards
            \<manufacturer>
                \<board name>
                    \<compiler>
                        \BSP
                            \*.*
```

`\Micrium`

Contains all software components and projects provided by Micrim.

`\Software`

This sub-directory contains all software components and projects.

`\EvalBoards`

This sub-directory contains all projects related to evaluation boards.

`\<manufacturer>`

The name of the manufacturer of the evaluation board. The “<” and “>” are not part of the actual name.

`\<board name>`

The name of the evaluation board. A board from Micrim will typically be called `uC-Eval-xxxx` where “`xxxx`” is the name of the CPU or MCU used on the evaluation board. The “<” and “>” are not part of the actual name.

`\<compiler>`

Copyright 2015 Micrium Inc.

The name of the compiler or compiler manufacturer used to build code for the evaluation board. The “<” and “>” are not part of the actual name.

\BSP

This directory is always called BSP.

.

The source files of the BSP. Typically all of the file names start with BSP. It is therefore normal to find `bsp.c` and `bsp.h` in this directory. BSP code should contain such functions as LED control functions, initialization of timers, interface to Ethernet controllers and more.

Copyright 2015 Micrium Inc.

uC-OS-III CPU Independent Source Code

The files in these directories are µC/OS-III processor independent files provided in source form.

```
\Micrium
    \Software
        \uCOS-III
            \Cfg\Template
                \os_app_hooks.c
                \os_cfg.h
                \os_cfg_app.h
            \Source
                \os_cfg_app.c
                \os_core.c
                \os_dbg.c
                \os_flag.c
                \os_int.c
                \os_mem.c
                \os_msg.c
                \os_mutex.c
                \os_pend_multi.c
                \os_prio.c
                \os_q.c
                \os_sem.c
                \os_stat.c
                \os_task.c
                \os_tick.c
                \os_time.c
                \os_tmr.c
                \OS_VAR
                \os.h
                \os_type.h
            \TLS
                \<tool>
                    \os_tls.c
```

\Micrium

Contains all software components and projects provided by Micriμm.

\Software

This sub-directory contains all software components and projects.

Copyright 2015 Micrium Inc.

`\uCOS-III`

This is the main µC/OS-III directory.

`\Cfg\Template`

This directory contains examples of configuration files to copy to the project directory. You will then modify these files to suit the needs of the application.

`os_app_hooks.c` shows how to write hook functions that are called by µC/OS-III. Specifically, this file contains eight empty functions.

`os_cfg.h` specifies which features of µC/OS-III are available for an application. The file is typically copied into an application directory and edited based on which features are required from µC/OS-III. See Appendix B, “µC/OS-III Configuration Manual”.

`os_cfg_app.h` is a configuration file that is typically copied into an application directory and edited based on application requirements. This file enables the user to determine the size of the idle task stack, the tick rate, the number of messages available in the message pool and more. See Appendix B, “µC/OS-III Configuration Manual”.

`\Source`

The directory containing the CPU-independent source code for µC/OS-III. All files in this directory should be included in the build. Features that are not required will be compiled out based on the value of `#define` constants in `os_cfg.h` and `os_cfg_app.h`.

`os_cfg_app.c` declares variables and arrays based on the values in `os_cfg_app.h`.

`os_core.c` contains core functionality for µC/OS-III such as `OSInit()` to initialize µC/OS-III, `OSSched()` for the task level scheduler, `OSIntExit()` for the interrupt level scheduler, pend list (or wait list) management (see Pend Lists), ready list management (see Ready List in General Statistics – Run-Time), and more.

`os_dbg.c` contains declarations of constant variables used by a kernel aware debugger or µC/Probe.

Copyright 2015 Micrium Inc.

`os_flag.c` contains the code for event flag management. See Synchronization for details about event flags.

`os_int.c` contains code for the interrupt handler task, which is used when `OS_CFG_ISR_POST_DEFERRED_EN` (see `os_cfg.h`) is set to `DEF_ENABLED`. See Interrupt Management for details regarding the interrupt handler task.

`os_mem.c` contains code for the µC/OS-III fixed-size memory manager, see Memory Management API Changes.

`os_msg.c` contains code to handle messages. µC/OS-III provides message queues and task specific message queues. `os_msg.c` provides common code for these two services. See Message Passing.

`os_mutex.c` contains code to manage mutual exclusion semaphores, see Resource Management.

`os_pend_multi.c` contains the code to allow code to pend on multiple semaphores or message queues. This is described in Pending On Multiple Objects.

`os_prio.c` contains the code to manage the bitmap table used to keep track of which tasks are ready-to-run, see The Ready List. This file can be replaced by an assembly language equivalent to improve performance if the CPU used provides bit set, clear and test instructions, and a count leading zeros instruction.

`os_q.c` contains code to manage message queues. See Message Passing.

`os_sem.c` contains code to manage semaphores used for resource management and/or synchronization. See Resource Management and Synchronization.

`os_stat.c` contains code for the statistic task, which is used to compute the global CPU usage and the CPU usage of each task. See About Task Management.

`os_task.c` contains code for managing tasks using `OSTaskCreate()`, `OSTaskDel()`, `OSTaskChangePrio()`, and many more. See About Task Management.

`os_tick.c` contains code to manage tasks that have delayed themselves or that are pending on a kernel object with a timeout. See About Task Management.

Copyright 2015 Micrium Inc.

`os_time.c` contains code to allow a task to delay itself until some time expires. See Time Management.

`os_tmr.c` contains code to manage software timers. See Timer Management.

`os_var.c` contains the µC/OS-III global variables. These variables are for µC/OS-III to manage and should not be accessed by application code.

`os.h` contains the main µC/OS-III header file, which declares constants, macros, µC/OS-III global variables (for use by µC/OS-III only), function prototypes, and more.

`os_type.h` contains declarations of µC/OS-III data types that can be changed by the port designer to make better use of the CPU architecture. In this case, the file would typically be copied to the port directory and then modified. See Appendix B, “µC/OS-III Configuration Manual”.

`\TLS\<tool>`

The directory containing the compiler interface functions that allow library functions to be thread safe. If this feature is not required then you can omit the files in this directory.

`os_tls.c` provides the compiler specific thread safe interface functions.

Copyright 2015 Micrium Inc.

uC-OS-III CPU Specific Source Code

The µC/OS-III port developer provides these files. See also Porting uC-OS-III.

```
\Micrium
    \Software
        \uCOS-III
            \Ports
                \<architecture>
                    \<compiler>
                        \os_cpu.h
                        \os_cpu_a.asm
                        \os_cpu_c.c
```

`\Micrium`

Contains all software components and projects provided by Micriμm.

`\Software`

This sub-directory contains all software components and projects.

`\uCOS-III`

The main µC/OS-III directory.

`\Ports`

The location of port files for the CPU architecture(s) to be used.

`\<architecture>`

This is the name of the CPU architecture that µC/OS-III was ported to. The “<” and “>” are not part of the actual name.

`\<compiler>`

The name of the compiler or compiler manufacturer used to build code for the port. The “<” and “>” are not part of the actual name.

Copyright 2015 Micrium Inc.

The files in this directory contain the µC/OS-III port, see Porting uC-OS-III for details on the contents of these files.

`os_cpu.h` contains a macro declaration for `OS_TASK_SW()`, as well as the function prototypes for at least the following functions: `OSCtxSw()`, `OSIntCtxSw()` and `OSStartHighRdy()`.

`os_cpu_a.asm` contains the assembly language functions to implement at least the following functions: `OSCtxSw()`, `OSIntCtxSw()` and `OSStartHighRdy()`.

`os_cpu_c.c` contains the C code for the port specific hook functions and code to initialize the stack frame for a task when the task is created.

Copyright 2015 Micrium Inc.

uC-CPU CPU Specific Source Code

µC/CPU consists of files that encapsulate common CPU-specific functionality and CPU and compiler-specific data types. See Porting uC-OS-III.

```
\Micrium
    \Software
        \uC-CPU
            \cpu_core.c
            \cpu_core.h
            \cpu_def.h
            \Cfg\Template
                \cpu_cfg.h
            \<architecture>
                \<compiler>
                    \cpu.h
                    \cpu_a.asm
                    \cpu_c.c
```

`\Micrium`

Contains all software components and projects provided by Micriµm.

`\Software`

This sub-directory contains all software components and projects.

`\uC-CPU`

This is the main µC/CPU directory.

`cpu_core.c` contains C code that is common to all CPU architectures. Specifically, this file contains functions to measure the interrupt disable time of the `CPU_CRITICAL_ENTER()` and `CPU_CRITICAL_EXIT()` macros, a function that emulates a count leading zeros instruction in case the CPU does not provide such an instruction, and a few other functions.

`cpu_core.h` contains function prototypes for the functions provided in `cpu_core.c` and allocation of the variables used by the module to measure interrupt disable time.

`cpu_def.h` contains miscellaneous `#define` constants used by the µC/CPU module.

Copyright 2015 Micrium Inc.

\Cfg\Template

This directory contains a configuration template file (cpu_cfg.h) that must be copied to the application directory to configure the µC/CPU module based on application requirements.

cpu_cfg.h determines whether to enable measurement of the interrupt disable time, whether the CPU implements a count leading zeros instruction in assembly language, or whether it will be emulated in C, and more.

\<architecture>

The name of the CPU architecture that µC/CPU was ported to. The "<" and ">" are not part of the actual name.

\<compiler>

The name of the compiler or compiler manufacturer used to build code for the µC/CPU port. The "<" and ">" are not part of the actual name.

The files in this directory contain the µC/CPU port, see Porting uC-OS-III for details on the contents of these files.

cpu.h contains type definitions to make µC/OS-III and other modules independent of the CPU and compiler word sizes. Specifically, one will find the declaration of the CPU_INT16U, CPU_INT32U, CPU_FP32 and many other data types. This file also specifies whether the CPU is a big or little endian machine, defines the CPU_STK data type used by µC/OS-III, defines the macros CPU_CRITICAL_ENTER() and CPU_CRITICAL_EXIT(), and contains function prototypes for functions specific to the CPU architecture, and more.

cpu_a.asm contains the assembly language functions to implement code to disable and enable CPU interrupts, count leading zeros (if the CPU supports that instruction), and other CPU specific functions that can only be written in assembly language. This file may also contain code to enable caches, setup MPUs and MMU, and more. The functions provided in this file are accessible from C.

cpu_c.c contains C code of functions that are based on a specific CPU architecture

Copyright 2015 Micrium Inc.

but written in C for portability. As a general rule, if a function can be written in C then it should be, unless there is significant performance benefits available by writing it in assembly language.

uC-LIB Portable Library Functions

µC/LIB consists of library functions meant to be highly portable and not tied to any specific compiler. This facilitates third-party certification of Micriµm products. µC/OS-III does not use any µC/LIB functions, however µC/OS-III and µC/CPU assumes the presence of `lib_def.h` for such definitions as: `DEF_YES`, `DEF_NO`, `DEF_TRUE`, `DEF_FALSE`, `DEF_ON`, `DEF_OFF` and more.

```
\Micrium
    \Software
        \uC-LIB
            \lib_ascii.c
            \lib_ascii.h
            \lib_def.h
            \lib_math.c
            \lib_math.h
            \lib_mem.c
            \lib_mem.h
            \lib_str.c
            \lib_str.h
            \Cfg\Template
                \lib_cfg.h
            \Ports
                \<architecture>
                    \<compiler>
                        \lib_mem_a.asm
```

`\Micrium`

Contains all software components and projects provided by Micrium.

`\Software`

This sub-directory contains all software components and projects.

`\uC-LIB`

This is the main µC/LIB directory.

`lib_ascii.c` and `lib_ascii.h` contain source code to replace some standard library functions such as `tolower()`, `toupper()`, `isalpha()`, `isdigit()`, etc. with µC/LIB equivalent functions `ASCII_ToLower()`, `ASCII_ToUpper()`, `ASCII_IsAlpha()`, and `ASCII_IsDig()`, respectively.

Copyright 2015 Micrium Inc.

lib_def.h defines constants for many common values such as TRUE/FALSE, YES/NO, ENABLED/DISABLED; as well as for integer, octet, and bit values. However, all #define in this file starts with DEF_ so those constants are actually called DEF_TRUE/DEF_FALSE, DEF_YES/DEF_NO, DEF_ENABLED/DEF_DISABLED, etc. This file also contains macros for common mathematical operations like min(), max(), abs(), bit_set(), bit_clr(), etc. with DEF_MIN(), DEF_MAX(), DEF_ABS(), DEF_BIT_SET(), DEF_BIT_CLR(), respectively.

lib_math.c and lib_math.h contain source code to replace some standard library functions such as rand(), srand(), etc. with µC/LIB equivalent functions Math_Rand(), Math_SetSeed(), respectively.

lib_mem.c and lib_mem.h contain source code to replace some standard library functions such as memclr(), memset(), memcpy(), memcmp(), etc. with µC/LIB equivalent functions Mem_Clr(), Mem_Set(), Mem_Copy(), Mem_Cmp(), respectively.

lib_str.c and lib_str.h contain source code to replace some standard library functions such as strlen(), strcpy(), strcmp(), memcmp(), etc. with µC/LIB equivalent functions Str_Lenr(), Str_Copy(), Str_Cmp(), respectively.

\Cfg\Template

This directory contains a configuration template file (lib_cfg.h) that should be copied to the application directory to configure the µC/LIB module based on application requirements.

lib_cfg.h determines whether to enable assembly language optimization (assuming there is an assembly language file for the processor, i.e., lib_mem_a.asm) and a few other #defines.

\Ports\Architecture\Compiler

This directory contains optimized assembly language files specific to the CPU architecture to replace C functions with much faster assembly language implementations. The presence of this folder depends on whether such assembly language functions were implemented by the port developer of the µC/LIB module.

lib_mem_a.asm contains optimized versions of the lib_mem.c functions.

Copyright 2015 Micrium Inc.

Copyright 2015 Micrium Inc.

Getting Started with uC-OS-III

µC/OS-III provides services to application code in the form of a set of functions that perform specific operations. µC/OS-III offers services to manage tasks, semaphores, message queues, mutual exclusion semaphores and more. As far as the application is concerned, it calls the µC/OS-III functions as if they were any other functions. In other words, the application now has access to a library of approximately 70 new functions.

In this section, the reader will appreciate how easy it is to start using µC/OS-III.

It is assumed that the project setup (files and directories) is as described in the previous chapter, and that a C compiler exists for the target processor that is in use. However, this chapter makes no assumptions about the tools or the processor that is used.

Copyright 2015 Micrium Inc.

Single Task Application

The listing below shows the top portion of a simple application file called app.c.

```
/*
*********************************************************************************************
*                                       INCLUDE FILES
*********************************************************************************************
*/
#include <app_cfg.h>                                                    (1)
#include <bsp.h>
#include <os.h>
/*
*********************************************************************************************
*                                    LOCAL GLOBAL VARIABLES
*********************************************************************************************
*/
static  OS_TCB           AppTaskStartTCB;                               (2)
static  CPU_STK          AppTaskStartStk[APP_TASK_START_STK_SIZE];      (3)
/*
*********************************************************************************************
*                                    FUNCTION PROTOTYPES
*********************************************************************************************
*/
static  void  AppTaskStart (void *p_arg);                               (4)
```

Listing - app.c (1st Part)

(1) As with any C programs, you need to include the necessary headers to build the application.

`app_cfg.h` is a header file that configures the application. For our example, `app_cfg.h` contains `#define` constants to establish task priorities, stack sizes, and other application specifics.

`bsp.h` is the header file for the Board Support Package (BSP), which defines macros (`#defines`) and function prototypes, such as `BSP_Init()`, `BSP_LED_On()`, `OS_TS_GET()` and more. The actual BSP might contain multiple header files. `bsp.h` is shown generically here.

`os.h` is the main header file for µC/OS-III, and includes the following header files:

```
os_cfg.h
cpu.h
cpu_core.h
lib_def.h
```

Copyright 2015 Micrium Inc.

```
os_type.h
os_cpu.h
```

(2) We will be creating an application task and it is necessary to allocate a task control block (`OS_TCB`) for this task. The `OS_TCB` data type will be described in About Task Management.

(3) Each task created requires its own stack. A stack must be declared using the `CPU_STK` data type as shown. The stack can be allocated statically as shown here, or dynamically from the heap using `malloc()`. It should not be necessary to free the stack space, because the task should never be destroyed, and thus, the stack would always be used.

(4) This is the function prototype of the task that we will create.

Most C applications start at `main()` as shown in in the listing below:

```
void  main (void)
{
    OS_ERR  err;

    BSP_IntDisAll();                                                        (1)
    OSInit(&err);                                                           (2)
    if (err != OS_ERR_NONE) {
        /* Something didn't get initialized correctly ...                  */
        /* ... check os.h for the meaning of the error code, see OS_ERR_xxxx */
    }
    OSTaskCreate((OS_TCB     *)&AppTaskStartTCB,                            (3)
                 (CPU_CHAR   *)"App Task Start",                            (4)
                 (OS_TASK_PTR )AppTaskStart,                                (5)
                 (void       *)0,                                           (6)
                 (OS_PRIO     )APP_TASK_START_PRIO,                         (7)
                 (CPU_STK    *)&AppTaskStartStk[0],                         (8)
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE / 10,                (9)
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,                    (10)
                 (OS_MSG_QTY  )0,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR), (11)
                 (OS_ERR     *)&err);                                       (12)
    if (err != OS_ERR_NONE) {
        /* The task didn't get created.  Lookup the value of the error code ... */
        /* ... in os.h for the meaning of the error                        */
    }
    OSStart(&err);                                                          (13)
    if (err != OS_ERR_NONE) {
        /* Your code is NEVER supposed to come back to this point.         */
    }
}
```

Listing - app.c (2nd Part)

Copyright 2015 Micrium Inc.

(1) The startup code for the compiler will bring the CPU to `main()`. `main()` then starts by calling a BSP function that disables all interrupts. On most processors, interrupts are disabled at startup until explicitly enabled by application code. However, it is safer to turn off all peripheral interrupts during startup.

(2) `OSInit()` is the called to initialize µC/OS-III. `OSInit()` initializes internal variables and data structures, and also creates between two (2) and five (5) internal tasks. Unless otherwise specified in the configuration file (`os_cfg.h`), at a minimum µC/OS-III creates the idle task (`OS_IdleTask()`), which executes when no other task is ready-to-run. µC/OS-III also creates the tick task, which is responsible for keeping track of time.

Depending on the value of `#define` constants, µC/OS-III will create the statistic task (`OS_StatTask()`), the timer task (`OS_TmrTask()`), and the interrupt handler queue management task (`OS_IntQTask()`). Those are all discussed in About Task Management.

Most of µC/OS-III's functions return an error code via a pointer to an `OS_ERR` variable, err in this case. If `OSInit()` was successful, err will be set to `OS_ERR_NONE`. If `OSInit()` encounters a problem during initialization, it will return immediately upon detecting the problem and set err accordingly. If this occurs, look up the error code value in `os.h`. Specifically, all error codes start with `OS_ERR_`.

It is important to note that `OSInit()` must be called before any other µC/OS-III function.

(3) You create a task by calling `OSTaskCreate()`. `OSTaskCreate()` requires 13 arguments. The first argument is the address of the `OS_TCB` that is declared for this task. About Task Management provides additional information about tasks.

(4) `OSTaskCreate()` allows a name to be assigned to each of the tasks. µC/OS-III stores a pointer to the task name inside the `OS_TCB` of the task. There is no limit on the number of ASCII characters used for the name.

(5) The third argument is the address of the task code. A typical µC/OS-III task is implemented as an infinite loop as shown:

Copyright 2015 Micrium Inc.

```
void  MyTask (void *p_arg)
         {
             /* Do something with “p_arg”.
             while (1) {
                 /* Task body */
             }
         }
```

Listing - app.c (3rd Part)

The task receives an argument when it first starts. As far as the task is concerned, it looks like any other C function that can be called by the code. However, your code *must not* call `MyTask()`. The call is actually performed through µC/OS-III.

(6) The fourth argument of `OSTaskCreate()` is the actual argument that the task receives when it first begins. In other words, the “`p_arg`” of `MyTask()`. In the example a `NULL` pointer is passed, and thus “`p_arg`” for `AppTaskStart()` will be a `NULL` pointer.

The argument passed to the task can actually be any pointer. For example, the user may pass a pointer to a data structure containing parameters for the task.

(7) The next argument to `OSTaskCreate()` is the priority of the task. The priority establishes the relative importance of this task with respect to the other tasks in the application. A low-priority number indicates a high priority (or more important task). You can set the priority of the task to any value between 1 and `OS_CFG_PRIO_MAX-2`, inclusively. Avoid using priority 0, and priority `OS_CFG_PRIO_MAX-1`, because these are reserved for µC/OS-III. `OS_CFG_PRIO_MAX` is a compile time configuration constant, which is declared in os_cfg.h.

(8) The sixth argument to `OSTaskCreate()` is the base address of the stack assigned to this task. The base address is always the lowest memory location of the stack.

(9) The next argument specifies the location of a “watermark” in the task’s stack that can be used to determine the allowable stack growth of the task. See About Task Management for more details on using this feature. In the code above, the value represents the amount of stack space (in `CPU_STK` elements) before the stack is empty. In other words, in the example, the limit is reached when there is 10% of the stack left.

(10) The eighth argument to `OSTaskCreate()` specifies the size of the task’s stack in number of `CPU_STK` elements (not bytes). For example, if you want to allocate 1 Kbyte of stack

Copyright 2015 Micrium Inc.

space for a task and the CPU_STK is a 32-bit word, then you need to pass 256.

(11) The next three arguments are skipped as they are not relevant for the current discussion. The 12th argument to OSTaskCreate() specifies options. In this example, we specify that the stack will be checked at run time (assuming the statistic task was enabled in os_cfg.h), and that the contents of the stack will be cleared when the task is created.

(12) The last argument of OSTaskCreate() is a pointer to a variable that will receive an error code. If OSTaskCreate() is successful, the error code will be OS_ERR_NONE otherwise, you can look up the value of the error code in os.h (see OS_ERR_xxxx) to determine the cause of the error.

(13) The final step in main() is to call OSStart(), which starts the multitasking process. Specifically, µC/OS-III will select the highest-priority task that was created before calling OSStart().

You should note that the highest-priority task is always OS_IntQTask() if that task is enabled in os_cfg.h (through the OS_CFG_ISR_POST_DEFERRED_EN constant). If this is the case, OS_IntQTask() will perform some initialization of its own and then µC/OS-III will switch to the next most important task that was created.

A few important points are worth noting. For one thing, you can create as many tasks as you want before calling OSStart(). However, it is recommended to only create one task as shown in the example because, having a single application task allows µC/OS-III to determine the relative speed of the CPU. This allows µC/OS-III to determine the percentage of CPU usage at runtime. Also, if the application needs other kernel objects such as semaphores and message queues then it is recommended that these be created prior to calling OSStart(). Finally, notice that interrupts are not enabled. This will be discussed next by examining the contents of AppTaskStart(), which is shown in the listing below:

Copyright 2015 Micrium Inc.

```
static  void  AppTaskStart (void  *p_arg)                       (1)
{
    OS_ERR  err;


    p_arg = p_arg;
    BSP_Init();                                                 (2)
    CPU_Init();                                                 (3)
    BSP_OS_TickInit();                                          (4)
    BSP_LED_Off(0);                                             (5)
    while (1) {                                                 (6)
        BSP_LED_Toggle(0);                                      (7)
        OSTimeDlyHMSM((CPU_INT16U)  0,                          (8)
                      (CPU_INT16U)  0,
                      (CPU_INT16U)  0,
                      (CPU_INT32U)100,
                      (OS_OPT    )OS_OPT_TIME_HMSM_STRICT,
                      (OS_ERR   *)&err);
        /* Check for ‘err’ */
    }
}
```

Listing - AppTaskStart()

(1) As previously mentioned, a task looks like any other C function. The argument “`p_arg`” is passed to `AppTaskStart()` by `OSTaskCreate()`, as discussed in the previous listing description.

(2) `BSP_Init()` is a Board Support Package (BSP) function that is responsible for initializing the hardware on an evaluation or target board. The evaluation board might have General Purpose Input Output (GPIO) lines that might need to be configured, relays, sensors and more. This functionality is found in a file called bsp.c.

(3) `CPU_Init()` initializes the µC/CPU services. µC/CPU provides services to measure interrupt latency, obtain time stamps, and provides emulation of the count leading zeros instruction if the processor used does not have that instruction, and more.

(4) `BSP_OS_TickInit()` sets up the µC/OS-III tick interrupt. For this, the function needs to initialize one of the hardware timers to interrupt the CPU at a rate of: `OSCfg_TickRate_Hz`, which is defined in `os_cfg_app.h` (See `OS_CFG_TICK_RATE_HZ`).

(5) `BSP_LED_Off()` is a function that will turn off all LEDs. `BSP_LED_Off()` is written such that a zero argument means all the LEDs.

(6) Most µC/OS-III tasks will need to be written as an infinite loop.

Copyright 2015 Micrium Inc.

(7) This BSP function toggles the state of the specified LED. Again, a zero indicates that all the LEDs should be toggled on the evaluation board. You simply change the zero to 1 and this will cause LED #1 to toggle. Exactly which LED is LED #1? That depends on the BSP developer. Specifically, access to LEDs are encapsulated through the functions: `BSP_LED_On()`, `BSP_LED_Off()` and `BSP_LED_Toggle()`. Also, for sake of portability, we prefer to assign LEDs logical values (1, 2, 3, etc.) instead of specifying which port and which bit on each port.

(8) Finally, each task in the application must call one of the µC/OS-III functions that will cause the task to "wait for an event". The task can wait for time to expire (by calling `OSTimeDly()`, or `OSTimeDlyHMSM()`), or wait for a signal or a message from an ISR or another task. In the code shown, we used `OSTimeDlyHMSM()` which allows a task to be suspended until the specified number of hours, minutes, seconds and milliseconds have expired. In this case, 100 ms. Time Management provides additional information about time delays.

Copyright 2015 Micrium Inc.

Multiple Tasks Application with Kernel Objects

The code of the listings below shows a more complete example and contains three tasks: a mutual exclusion, semaphore, and a message queue.

```
/*
*********************************************************************************************************
*                                          INCLUDE FILES
*********************************************************************************************************
*/
#include <app_cfg.h>
#include <bsp.h>
#include <os.h>
/*
*********************************************************************************************************
*                                      LOCAL GLOBAL VARIABLES
*********************************************************************************************************
*/
static  OS_TCB           AppTaskStartTCB;                               (1)
static  OS_TCB           AppTask1_TCB;
static  OS_TCB           AppTask2_TCB;
static  OS_MUTEX         AppMutex;                                      (2)
static  OS_Q             AppQ;                                          (3)
static  CPU_STK          AppTaskStartStk[APP_TASK_START_STK_SIZE];      (4)
static  CPU_STK          AppTask1_Stk[128];
static  CPU_STK          AppTask2_Stk[128];
/*
*********************************************************************************************************
*                                       FUNCTION PROTOTYPES
*********************************************************************************************************
*/
static  void  AppTaskStart (void *p_arg);                               (5)
static  void  AppTask1     (void *p_arg);
static  void  AppTask2     (void *p_arg);
```

Listing - app.c (1st Part)

(1) Here we allocate storage for the `OS_TCB`s of each task.

(2) A mutual exclusion semaphore (a.k.a. a mutex) is a kernel object (a data structure) that is used to protect a shared resource from being accessed by more than one task. A task that wants to access the shared resource must obtain the mutex before it is allowed to proceed. The owner of the resource relinquishes the mutex when it has finished accessing the resource. This process is demonstrated in this example.

(3) A message queue is a kernel object through which Interrupt Service Routines (ISRs) and/or tasks send messages to other tasks. The sender “formulates” a message and sends it to the message queue. The task(s) wanting to receive these messages wait on the

Copyright 2015 Micrium Inc.

message queue for messages to arrive. If there are already messages in the message queue, the receiver immediately retrieves those messages. If there are no messages waiting in the message queue, then the receiver will be placed in a wait list associated with the message queue. This process will be demonstrated in this example.

(4) A stack is allocated for each task.

(5) The prototype of the tasks are declared.

The listing below shows the C entry point, i.e. `main()` .

```
void  main (void)
{
    OS_ERR  err;

    BSP_IntDisAll();
    OSInit(&err);
    /* Check for ‘err’ */

    OSMutexCreate((OS_MUTEX  *)&AppMutex,                                          (1)
                  (CPU_CHAR   *)”My App. Mutex”,
                  (OS_ERR     *)&err);
    /* Check for ‘err’ */

    OSQCreate    ((OS_Q      *)&AppQ,                                              (2)
                  (CPU_CHAR   *)”My App Queue”,
                  (OS_MSG_QTY  )10,
                  (OS_ERR     *)&err);
    /* Check for ‘err’ */

    OSTaskCreate((OS_TCB     *)&AppTaskStartTCB,                                   (3)
                 (CPU_CHAR   *)”App Task Start”,
                 (OS_TASK_PTR )AppTaskStart,
                 (void       *)0,
                 (OS_PRIO     )APP_TASK_START_PRIO,
                 (CPU_STK    *)&AppTaskStartStk[0],
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE / 10,
                 (CPU_STK_SIZE)APP_TASK_START_STK_SIZE,
                 (OS_MSG_QTY  )0,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR     *)&err);
   /* Check for ‘err’ */

    OSStart(&err);
   /* Check for ‘err’ */
}
```

Listing - app.c (2nd Part)

(1) Creating a mutex is simply a matter of calling `OSMutexCreate()`. You need to specify the address of the `OS_MUTEX` object that will be used for the mutex. Resource Management provides additional information about mutual exclusion semaphores.

You can assign an ASCII name to the mutex, which is useful when debugging.

(2) You create the message queue by calling `OSQCreate()` and specify the address of the `OS_Q` object. Message Passing provides additional information about message queues.

You can assign an ASCII name to the message queue which can also be useful during debugging.

You need to specify how many messages the message queue is allowed to receive. This value must be greater than zero. If the sender sends messages faster than they can be consumed by the receiving task, messages will be lost. This can be corrected by either increasing the size of the message queue, or increasing the priority of the receiving task.

(3) The first application task is created.

The listing below shows how to create other tasks once multitasking as started.

Copyright 2015 Micrium Inc.

```
static  void  AppTaskStart (void *p_arg)
{
    OS_ERR  err;


    p_arg = p_arg;
    BSP_Init();
    CPU_Init();
    BSP_OS_TickInit();
    OSTaskCreate((OS_TCB     *)&AppTask1_TCB,                                    (1)
                 (CPU_CHAR   *)"App Task 1",
                 (OS_TASK_PTR )AppTask1,
                 (void       *)0,
                 (OS_PRIO     )5,
                 (CPU_STK    *)&AppTask1_Stk[0],
                 (CPU_STK_SIZE)0,
                 (CPU_STK_SIZE)128,
                 (OS_MSG_QTY  )0,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR     *)&err);

    OSTaskCreate((OS_TCB     *)&AppTask2_TCB,                                    (2)
                 (CPU_CHAR   *)"App Task 2",
                 (OS_TASK_PTR )AppTask2,
                 (void       *)0,
                 (OS_PRIO     )6,
                 (CPU_STK    *)&AppTask2_Stk[0],
                 (CPU_STK_SIZE)0,
                 (CPU_STK_SIZE)128,
                 (OS_MSG_QTY  )0,
                 (OS_TICK     )0,
                 (void       *)0,
                 (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
                 (OS_ERR     *)&err);
    BSP_LED_Off(0);
    while (1) {
        BSP_LED_Toggle(0);
        OSTimeDlyHMSM((CPU_INT16U)  0,
                      (CPU_INT16U)  0,
                      (CPU_INT16U)  0,
                      (CPU_INT32U)100,
                      (OS_OPT    )OS_OPT_TIME_HMSM_STRICT,
                      (OS_ERR   *)&err);
    }
}
```

Listing - app.c (3rd Part)

(1) Task #1 is created by calling `OSTaskCreate()`. If this task happens to have a higher priority than the task that creates it, µC/OS-III will immediately start Task #1. If the created task has a lower priority, `OSTaskCreate()` will return to `AppTaskStart()` and continue execution.

(2) Task #2 is created and if it has a higher priority than `AppTaskStart()`, µC/OS-III will immediately switch to that task.

```
static  void  AppTask1 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;


    p_arg = p_arg;
    while (1) {
        OSTimeDly ((OS_TICK     )1,                                    (1)
                   (OS_OPT      )OS_OPT_TIME_DLY,
                   (OS_ERR     *)&err);
        OSQPost    ((OS_Q      *)&AppQ,                               (2)
                    (void      *)1;
                    (OS_MSG_SIZE)sizeof(void *),
                    (OS_OPT     )OS_OPT_POST_FIFO,
                    (OS_ERR    *)&err);
        OSMutexPend((OS_MUTEX  *)&AppMutex,                           (3)
                    (OS_TICK    )0,
                    (OS_OPT     )OS_OPT_PEND_BLOCKING;
                    (CPU_TS    *)&ts,
                    (OS_ERR    *)&err);
        /* Access shared resource */                                  (4)
        OSMutexPost((OS_MUTEX  *)&AppMutex,                           (5)
                    (OS_OPT     )OS_OPT_POST_NONE,
                    (OS_ERR    *)&err);
    }
}
```

Listing - app.c (4th Part)

(1) The task starts by waiting for one tick to expire before it does anything useful. If the µC/OS-III tick rate is configured for 1000 Hz, the task will be suspended for 1 millisecond.

(2) The task then sends a message to another task using the message queue `AppQ`. In this case, the example sends a fixed message of value “1,” but the message could have consisted of the address of a buffer, the address of a function, or whatever would need to be sent.

(3) The task then waits on the mutual exclusion semaphore since it needs to access a shared resource with another task. If the resource is already owned by another task, `AppTask1()` will wait forever for the mutex to be released by its current owner. The forever wait is specified by passing 0 as the second argument of the call.

(4) When `OSMutexPend()` returns, the task owns the resource and can therefore access the shared resource. The shared resource may be a variable, an array, a data structure, an I/O

Copyright 2015 Micrium Inc.

device, etc. You should note that we didn't actually show the access to the shared resource. This is not relevant at this point.

(5) When the task is done with the shared resource, it must call `OSMutexPost()` to release the mutex.

```
static  void  AppTask2 (void *p_arg)
{
    OS_ERR       err;
    void        *p_msg;
    OS_MSG_SIZE  msg_size;
    CPU_TS       ts;
    CPU_TS       ts_delta;


    p_arg = p_arg;
    while (1) {
        p_msg = OSQPend((OS_Q        *)&AppQ,                              (1)
                        (OS_MSG_SIZE *)&msg_size,
                        (OS_TICK      )0,
                        (OS_OPT       )OS_OPT_PEND_BLOCKING,
                        (CPU_TS      *)&ts,
                        (OS_ERR      *)&err);
        ts_delta = OS_TS_GET() – ts;                                      (2)
        /* Process message received */                                    (3)
    }
}
```

Listing - app.c (5th Part)

(1) Task #2 starts by waiting for messages to be sent through the message queue `AppQ`. The task waits forever for a message to be received because the third argument specifies an infinite timeout.

When the message is received `p_msg` will contain the message (i.e., a pointer to "something"). In our case, `AppTask2()` will always receive a message value of '`1`'. Both the sender and receiver must agree as to the meaning of the message. The size of the message received is saved in "`msg_size`". Note that "`p_msg`" could point to a buffer and "`msg_size`" would indicate the size of this buffer.

Also, when the message is received, "`ts`" will contain the timestamp of when the message was sent. A timestamp is the value read from a fairly fast free-running timer. The timestamp is typically an unsigned 32-bit (or more) value.

Copyright 2015 Micrium Inc.

(2) Knowing when the message was sent allows the user to determine how long it took this task to get the message. This is done by reading the current timestamp and subtracting the timestamp of when the message was sent allows users to know how long it took for the message to be received. Note that the receiving task may not get the message immediately since ISRs or other higher-priority tasks might execute before the receiver gets to run.

(3) Here you would add your own code to process the received message.

Copyright 2015 Micrium Inc.

Critical Sections

A critical section of code, also called a *critical region*, is code that needs to be treated indivisibly. There are many critical sections of code contained in µC/OS-III. If a critical section is accessible by an Interrupt Service Routine (ISR) and a task, then disabling interrupts is necessary to protect the critical region. If the critical section is only accessible by task level code, the critical section may be protected through the use of a *preemption lock.*

Within µC/OS-III, the critical section access method depends on which ISR post method is used by interrupts (see Interrupt Management). If `OS_CFG_ISR_POST_DEFERRED_EN` is set to `DEF_DISABLED` (see `os_cfg.h`) then µC/OS-III will disable interrupts when accessing internal critical sections. If `OS_CFG_ISR_POST_DEFERRED_EN` is set to `DEF_ENABLED` then µC/OS-III will lock the scheduler when accessing most of its internal critical sections.

Interrupt Management discusses how to select the method to use.

µC/OS-III defines one macro for entering a critical section and two macros for leaving:

```
OS_CRITICAL_ENTER(),
OS_CRITICAL_EXIT() and
OS_CRITICAL_EXIT_NO_SCHED()
```

These macros are internal to µC/OS-III and must not be invoked by the application code. However, if you need to access critical sections in your application code, consult Resource Management.

Copyright 2015 Micrium Inc.

Disabling Interrupts

When setting OS_CFG_ISR_POST_DEFERRED_EN to DEF_DISABLED, µC/OS-III will disable interrupts before entering a critical section and re-enable them when leaving the critical section.

OS_CRITICAL_ENTER() invokes the µC/CPU macro CPU_CRITICAL_ENTER() that, in turn, calls CPU_SR_Save(). CPU_SR_Save() is a function typically written in assembly language that saves the current interrupt disable status and then disables interrupts. The saved interrupt disable status is returned to the caller and in fact, it is stored onto the caller's stack in a variable called "cpu_sr".

OS_CRITICAL_EXIT() and OS_CRITICAL_EXIT_NO_SCHED() both invoke the µC/CPU macro CPU_CRITICAL_EXIT(), which maps to CPU_SR_Restore(). CPU_SR_Restore() is passed the value of the saved "cpu_sr" variable to re-establish interrupts the way they were prior to calling OS_CRITICAL_ENTER().

The typical code for the macros is shown in the listing below:

```
#define  OS_CRITICAL_ENTER()          { CPU_CRITICAL_ENTER(); }
#define  OS_CRITICAL_EXIT()           { CPU_CRITICAL_EXIT();  }
#define  OS_CRITICAL_EXIT_NO_SCHED()  { CPU_CRITICAL_EXIT();  }
```

Listing - Critical section code – Disabling interrupts

Measuring Interrupt Disable Time

µC/CPU provides facilities to measure the amount of time interrupts are disabled. This is done by setting the configuration constant `CPU_CFG_INT_DIS_MEAS_EN` to 1 in `cpu_cfg.h`.

The measurement is started each time interrupts are disabled and ends when interrupts are re-enabled. The measurement keeps track of two values: a global interrupt disable time, and an interrupt disable time for each task. Therefore, it is possible to know how long a task disables interrupts, enabling the user to better optimize their code.

The per-task interrupt disable time is saved in the task's `OS_TCB` during a context switch (see `OSTaskSwHook()` in `os_cpu_c.c` and described in Context Switching).

The unit of measure for the measured time is in CPU_TS (timestamp) units. It is necessary to find out the resolution of the timer used to measure these timestamps. For example, if the timer used for the timestamp is incremented at 1 MHz then the resolution of `CPU_TS` is 1 microsecond.

Measuring the interrupt disable time obviously adds measurement artifacts and thus increases the amount of time the interrupts are disabled. However, as far as the measurement is concerned, measurement overhead is accounted for and the measured value represents the actual interrupt disable time as if the measurement was not present.

Interrupt disable time is obviously greatly affected by the speed at which the processor accesses instructions and thus, the memory access speed. In this case, the hardware designer might have introduced wait states to memory accesses, which affects overall performance of the system. This may show up as unusually long interrupt disable times.

Copyright 2015 Micrium Inc.

Locking the Scheduler

When setting `OS_CFG_ISR_POST_DEFERRED_EN` to `DEF_ENABLED`, µC/OS-III locks the scheduler before entering a critical section and unlocks the scheduler when leaving the critical section.

`OS_CRITICAL_ENTER()` simply increments `OSSchedLockNestingCtr` to lock the scheduler. This is the variable the scheduler uses to determine whether or not the scheduler is locked. It is locked when the value is non-zero.

`OS_CRITICAL_EXIT()` decrements `OSSchedLockNestingCtr` and when the value reaches zero, invokes the scheduler.

`OS_CRITICAL_EXIT_NO_SCHED()` also decrements `OSSchedLockNestingCtr`, but does not invoke the scheduler when the value reaches zero.

The code for the macros is shown in the listing below:

```
#define  OS_CRITICAL_ENTER()          {                                                \
                                          CPU_CRITICAL_ENTER();                       \
                                          OSSchedLockNestingCtr++;                    \
                                          CPU_CRITICAL_EXIT();                        \
                                      }
#define  OS_CRITICAL_EXIT()           {
                                          CPU_CRITICAL_ENTER();                       \
                                          OSSchedLockNestingCtr--;                    \
                                          if (OSSchedLockNestingCtr == (OS_NESTING_CTR)0) {  \
                                              CPU_CRITICAL_EXIT();                    \
                                              OSSched();                              \
                                          } else {                                    \
                                              CPU_CRITICAL_EXIT();                    \
                                          }                                           \
                                      }
#define  OS_CRITICAL_EXIT_NO_SCHED()  {
                                          CPU_CRITICAL_ENTER();                       \
                                          OSSchedLockNestingCtr--;                    \
                                          CPU_CRITICAL_EXIT();                        \
                                      }
```

Listing - Critical section code – Locking the Scheduler

Copyright 2015 Micrium Inc.

Measuring Scheduler Lock Time

µC/OS-III provides facilities to measure the amount of time the scheduler is locked. This is done by setting the configuration constant `OS_CFG_SCHED_LOCK_TIME_MEAS_EN` to `DEF_ENABLED` in `os_cfg.h`.

The measurement is started each time the scheduler is locked and ends when the scheduler is unlocked. The measurement keeps track of two values: a global scheduler lock time, and a per-task scheduler lock time. It is therefore possible to know how long each task locks the scheduler allowing the user to better optimize code.

The per-task scheduler lock time is saved in the task's `OS_TCB` during a context switch (see `OSTaskSwHook()` in `os_cpu_c.c` and described in Context Switching).

The unit of measure for the measured time is in `CPU_TS` (timestamp) units so it is necessary to find the resolution of the timer used to measure the timestamps. For example, if the timer used for the timestamp is incremented at 1 MHz then the resolution of `CPU_TS` is 1 microsecond.

Measuring the scheduler lock time adds measurement artifacts and thus increases the amount of time the scheduler is actually locked. However, measurement overhead is accounted for and the measured value represents the actual scheduler lock time as if the measurement was not present.

Copyright 2015 Micrium Inc.

uC-OS-III Features with Longer Critical Sections

The table below shows several µC/OS-III features that have potentially longer critical sections. Knowledge of these will help the user decide whether to direct µC/OS-III to use one critical section over another.

Feature	Reason
Multiple tasks at the same priority	Although this is an important feature of µC/OS-III, multiple tasks at the same priority create longer critical sections. However, if there are only a few tasks at the same priority, interrupt latency would be relatively small. If multiple tasks are not created at the same priority, use the interrupt disable method.
Event Flags See Synchronization	If multiple tasks are waiting on different events, going through all of the tasks waiting for events requires a fair amount of processing time, which means longer critical sections. If only a few tasks (approximately one to five) are waiting on an event flag group, the critical section would be short enough to use the interrupt disable method.
Pend on multiple objects See Pending on Multiple Objects	Pending on multiple objects is probably the most complex feature provided by µC/OS-III, requiring interrupts to be disabled for fairly long periods of time should the interrupt disable method be selected. If pending on multiple objects, it is highly recommended that the user select the scheduler-lock method. If the application does not use this feature, the interrupt disable method is an alternative.
Broadcast on Post calls See `OSSemPost()` and `OSQPost()` descriptions µC/OS-III API Reference.	µC/OS-III disables interrupts while processing a post to multiple tasks in a broadcast. When not using the broadcast option, you can use the interrupt disable method.

Table - Disabling interrupts or locking the Scheduler

Copyright 2015 Micrium Inc.

About Task Management

The design process of a real-time application generally involves splitting the work to be completed into tasks, each responsible for a portion of the problem. µC/OS-III makes it easy for an application programmer to adopt this paradigm. A task (also called a *thread*) is a simple program that thinks it has the Central Processing Unit (CPU) all to itself. On a single CPU, only one task can execute at any given time.

µC/OS-III supports multitasking and allows the application to have any number of tasks. The maximum number of task is actually only limited by the amount of memory (both code and data space) available to the processor. Multitasking is the process of *scheduling* and *switching* the CPU between several tasks (this will be expanded upon later). The CPU switches its attention between several *sequential* tasks. Multitasking provides the illusion of having multiple CPUs and, actually maximizes the use of the CPU. Multitasking also helps in the creation of modular applications. One of the most important aspects of multitasking is that it allows the application programmer to manage the complexity inherent in real-time applications. Application programs are typically easier to design and maintain when multitasking is used.

Tasks are used for such chores as monitoring inputs, updating outputs, performing computations, control loops, update one or more displays, reading buttons and keyboards, communicating with other systems, and more. One application may contain a handful of tasks while another application may require hundreds. The number of tasks does not establish how good or effective a design may be, it really depends on what the application (or product) needs to do. The amount of work a task performs also depends on the application. One task may have a few microseconds worth of work to perform while another task may require tens of milliseconds.

Tasks look like just any other C function, with a few small differences. There are two types of tasks: run-to-completion (the first listing below) and infinite loop (the second listing below). In most embedded systems, tasks typically take the form of an infinite loop. Also, no task is allowed to return as other C functions can. Given that a task is a regular C function, it can declare local variables.

When a µC/OS-III task begins executing, it is passed an argument, `p_arg`. This argument is a pointer to a void. The pointer is a universal vehicle used to pass your task the address of a variable, a structure, or even the address of a function, if necessary. With this pointer, it is

Copyright 2015 Micrium Inc.

possible to create many identical tasks, that all use the same code (or task body), but, with different run-time characteristics. For example, you may have four asynchronous serial ports that are each managed by their own task. However, the task code is actually identical. Instead of copying the code four times, you can create the code for a “generic” task that receives a pointer to a data structure, which contains the serial port’s parameters (baud rate, I/O port addresses, interrupt vector number, etc.) as an argument. In other words, you can instantiate the same task code four times and pass it different data for each serial port that each instance will manage.

A run-to-completion task must `delete` itself by calling `OSTaskDel()`. The task starts, performs its function, and terminates. There would typically not be too many such tasks in the embedded system because of the overhead associated with “creating” and “deleting” tasks at run-time. In the task body, you can call most of µC/OS-III’s functions to help perform the desired operation of the task.

```
void MyTask (void *p_arg)
{
    OS_ERR err;
    /* Local variables                    */



    /* Do something with ‘p_arg’          */
    /* Task initialization                */
    /* Task body ... do work!             */
    OSTaskDel((OS_TCB *)0, &err);
}
```

Listing - Run-To-Completion task

With µC/OS-III, you either can call a C or assembly language functions from a task. In fact, it is possible to call the same C function from different tasks as long as the functions are reentrant. A *reentrant* function is a function that does not use static or otherwise global variables unless they are protected (µC/OS-III provides mechanisms for this) from multiple access. If shared C functions only use local variables, they are generally reentrant (assuming that the compiler generates reentrant code). An example of a non-reentrant function is the famous strtok() provided by most C compilers as part of the standard library. This function is used to parse an ASCII string for “tokens.” The first time you call this function, you specify the ASCII string to parse and a list of token delimiters. As soon as the function finds the first token, it returns. The function “remembers” where it was last so when called again, it can extract additional tokens, which is clearly non-reentrant.

The use of an infinite loop is more common in embedded systems because of the repetitive

Copyright 2015 Micrium Inc.

work needed in such systems (reading inputs, updating displays, performing control operations, etc.). This is one aspect that makes a task different than a regular C function. Note that one could use a "`while (1)`" or "`for (;;)`" to implement the infinite loop, since both behave the same. The one used is simply a matter of personal preference. At Micrium, we like to use "`while (DEF_ON)`". The infinite loop *must* call a µC/OS-III service (i.e., function) that will cause the task to wait for an event to occur. It is important that each task wait for an event to occur, otherwise the task would be a true infinite loop and there would be no easy way for other lower priority tasks to execute. This concept will become clear as more is understood regarding µC/OS-III.

```
void MyTask (void *p_arg)
{
    /* Local variables                                          */

    /* Do something with "p_arg"                                */
    /* Task initialization                                      */
    while (DEF_ON) {      /* Task body, as an infinite loop.    */
        :
        /* Task body ... do work!                               */
        :
        /* Must call one of the following services:             */
        /*    OSFlagPend()                                      */
        /*    OSMutexPend()                                     */
        /*    OSPendMulti()                                     */
        /*    OSQPend()                                         */
        /*    OSSemPend()                                       */
        /*    OSTimeDly()                                       */
        /*    OSTimeDlyHMSM()                                   */
        /*    OSTaskQPend()                                     */
        /*    OSTaskSemPend()                                   */
        /*    OSTaskSuspend()     (Suspend self)                */
        /*    OSTaskDel()         (Delete  self)                */
        :
        /* Task body ... do work!                               */
        :
    }
}
```

Listing - Infinite Loop task

The event the task is waiting for may simply be the passage of time (when `OSTimeDly()` or `OSTimeDlyHMSM()` is called). For example, a design may need to scan a keyboard every 100 milliseconds. In this case, you would simply delay the task for 100 milliseconds then see if a key was pressed on the keyboard and, possibly perform some action based on which key was pressed. Typically, however, a keyboard scanning task should just buffer an "identifier" unique to the key pressed and use another task to decide what to do with the key(s) pressed.

Similarly, the event the task is waiting for could be the arrival of a packet from an Ethernet controller. In this case, the task would call one of the `OS???Pend()` calls (pend is synonymous

Copyright 2015 Micrium Inc.

with wait). The task will have nothing to do until the packet is received. Once the packet is received, the task processes the contents of the packet, and possibly moves the packet along a network stack.

It’s important to note that when a task waits for an event, it does not consume CPU time.

Tasks must be created in order for µC/OS-III to know about tasks. You create a task by simply calling `OSTaskCreate()` as we’ve seen in the Getting Started section. The function prototype for OSTaskCreate() is shown below:

```
void OSTaskCreate  (OS_TCB         *p_tcb,
                    OS_CHAR        *p_name,
                    OS_TASK_PTR     p_task,
                    void           *p_arg,
                    OS_PRIO         prio,
                    CPU_STK        *p_stk_base,
                    CPU_STK_SIZE    stk_limit,
                    CPU_STK_SIZE    stk_size,
                    OS_MSG_QTY      q_size,
                    OS_TICK         time_slice,
                    void           *p_ext,
                    OS_OPT          opt,
                    OS_ERR         *p_err)
```

Listing - OSTaskCreate()

A complete description of `OSTaskCreate()` and its arguments is provided in uC-OS-III API Reference. However, it is important to understand that a task needs to be assigned a *Task Control Block* (i.e., TCB), a stack, a priority and a few other parameters which are initialized by `OSTaskCreate()`, as shown in the figure below.

Copyright 2015 Micrium Inc.

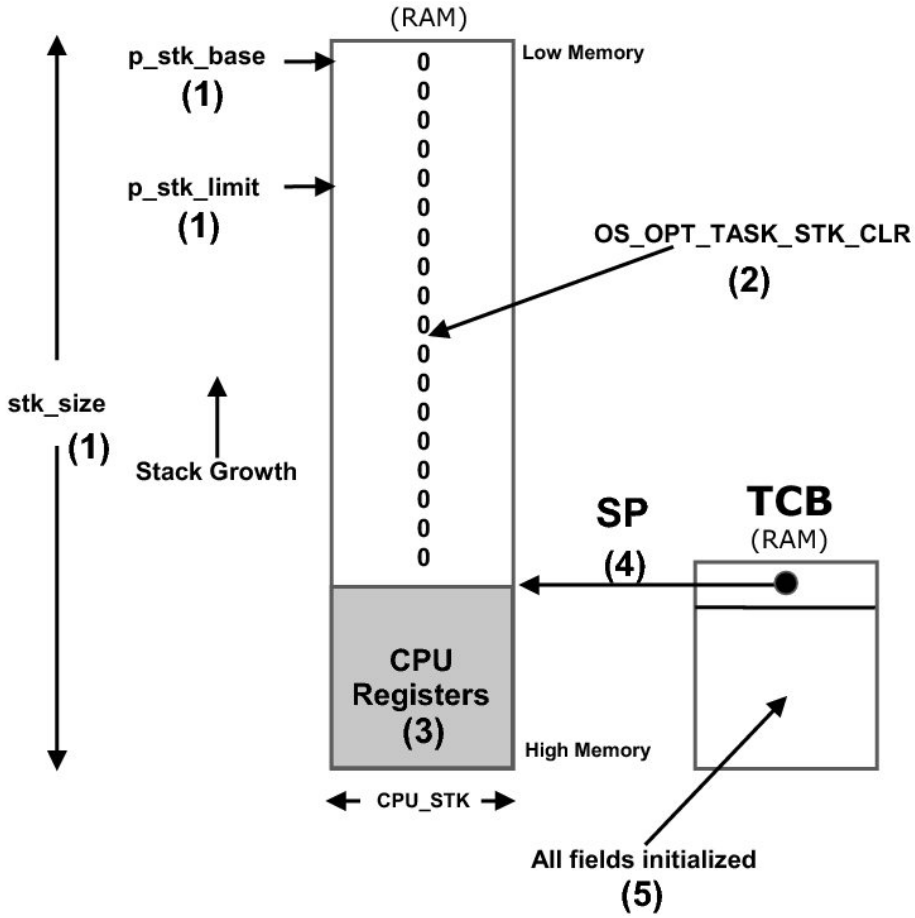


Figure - OSTaskCreate() initializes the task's TCB and stack

(1) When calling `OSTaskCreate()`, you pass the base address of the stack (`p_stk_base`) that will be used by the task, the watermark limit for stack growth (stk_limit) which is expressed in number of `CPU_STK` entries before the stack is empty, and the size of that stack (`stk_size`), also in number of `CPU_STK` elements.

(2) When specifying `OS_OPT_TASK_STK_CHK + OS_OPT_TASK_STK_CLR` in the opt argument of `OSTaskCreate()`, µC/OS-III initializes the task's stack with all zeros.

(3) µC/OS-III then initializes the top of the task's stack with a copy of the CPU registers in the same stacking order as if they were all saved at the beginning of an ISR. This makes it easy to perform context switches as we will see when discussing the context switching

Copyright 2015 Micrium Inc.

process. For illustration purposes, the assumption is that the stack grows from high memory to low memory, but the same concept applies for CPUs that use the stack in the reverse order.

(4) The new value of the stack pointer (SP) is saved in the TCB. Note that this is also called the `top-of-stack`.

(5) The remaining fields of the TCB are initialized: task priority, task name, task state, internal message queue, internal semaphore, and many others.

Next, a call is made to a function that is defined in the CPU port, `OSTaskCreateHook()` (see `os_cpu_c.c`). `OSTaskCreateHook()` is passed the pointer to the new TCB and this function allows you (or the port designer) to extend the functionality of `OSTaskCreate()`. For example, one could printout the contents of the fields of the newly created TCB onto a terminal for debugging purposes.

The task is then placed in the ready-list (see The Ready List) and finally, if multitasking has started, µC/OS-III will invoke the scheduler to see if the created task is now the highest priority task and, if so, will context switch to this new task.

The body of the task can invoke other services provided by µC/OS-III. Specifically, a task can create another task (i.e., call `OSTaskCreate()`), suspend and resume other tasks (i.e., call `OSTaskSuspend()` and `OSTaskResume()` respectively), post signals or messages to other tasks (i.e., call `OS??Post()`), share resources with other tasks, and more. In other words, tasks are not limited to only make “wait for an event” function calls.

The figure below shows the resources with which a task typically interacts.

Copyright 2015 Micrium Inc.

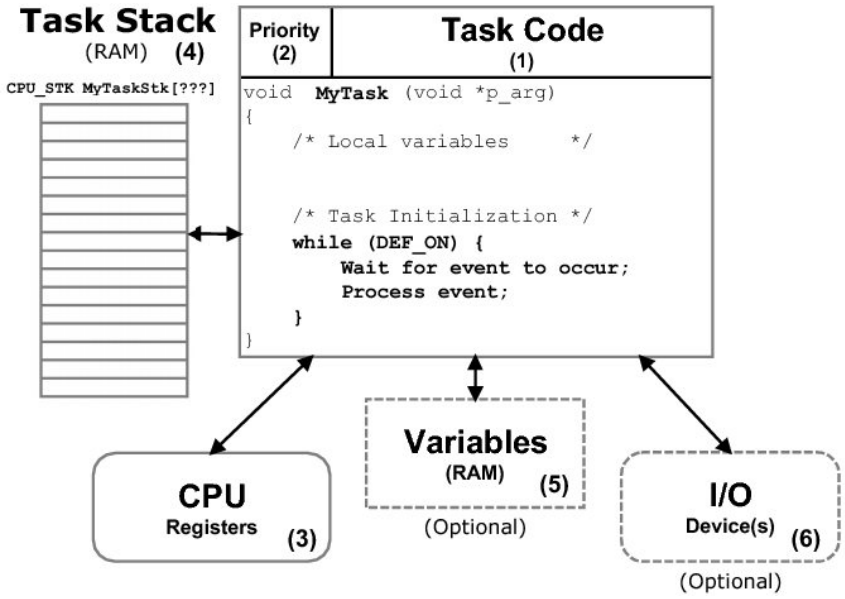


Figure - Tasks interact with resources

(1) An important aspect of a task is its code. As previously mentioned, the code looks like any other C function, except that it is typically implemented as an infinite loop and, a task is not allowed to return.

(2) Each task is assigned a priority based on its importance in the application. µC/OS-III's job is to decide which task will run on the CPU. The general rule is that µC/OS-III will run the most important `ready-to-run` task (highest priority).

With µC/OS-III, a low priority number indicates a high priority. In other words, a task at priority 1 is more important than a task at priority 10.

µC/OS-III supports a compile-time user configurable number of different priorities (see `OS_PRIO_MAX` in `os_cfg.h`). Thus, µC/OS-III allows the user to determine the number of different priority levels the application is allowed to use. Also, µC/OS-III supports an unlimited number of tasks at the same priority. For example, µC/OS-III can be configured to have 64 different priority levels and one can assign dozens of tasks at each priority level.

See Assigning Task Priorities.

Copyright 2015 Micrium Inc.

(3) A task has its own set of CPU registers. As far as a task is concerned, the task thinks it actually has the CPU all to itself.

(4) Because µC/OS-III is a preemptive kernel, each task must have its own stack area. The stack always resides in RAM and is used to keep track of local variables, function calls, and possibly ISR (Interrupt Service Routine) nesting.

Stack space can be allocated either statically (at compile-time) or dynamically (at run-time). A static stack declaration is shown below. This declaration is made outside of a function.

```
static CPU_STK MyTaskStk[???];
```

or,

```
CPU_STK MyTaskStk[???];
```

Note that "`???`" indicates that the size of the stack (and thus the array) depends on the task stack requirements. Stack space may be allocated dynamically by using the C compiler's heap management function (i.e., `malloc()`) as shown below. However, care must be taken with fragmentation. If creating and deleting tasks, the process of allocating memory might not be able to provide a stack for the task(s) because the heap will eventually become fragmented. For this reason, allocating stack space dynamically in an embedded system is typically allowed but, once allocated, stacks should not be deallocated. Said another way, it's fine to create a task's stack from the heap as long as you don't free the stack space back to the heap.

Copyright 2015 Micrium Inc.

Assigning Task Priorities

Sometimes task priorities are both obvious and intuitive. For example, if the most important aspect of the embedded system is to perform some type of control and it is known that the control algorithm must be responsive then it is best to assign the control task(s) a high priority while display and operator interface tasks are assigned low priority. However, most of the time, assigning task priorities is not so cut and dry because of the complex nature of real-time systems. In most systems, not all tasks are considered critical, and non-critical tasks should obviously be given low priorities.

An interesting technique called rate monotonic scheduling (RMS) assigns task priorities based on how often tasks execute. Simply put, tasks with the highest rate of execution are given the highest priority. However, RMS makes a number of assumptions, including:

- All tasks are periodic (they occur at regular intervals).
- Tasks do not synchronize with one another, share resources, or exchange data.
- The CPU must always execute the highest priority task that is ready-to-run. In other words, preemptive scheduling must be used.

Given a set of `n` tasks that are assigned RMS priorities, the basic RMS theorem states that all task hard real-time deadlines are always met if the following inequality holds true:

$$\sum_i \frac{E_i}{T_i} \leq n\left(2^{1/n}-1\right)$$

Where `Ei` corresponds to the maximum execution time of task `i`, and `Ti` corresponds to the execution period of task `i`. In other words, `Ei/Ti` corresponds to the fraction of CPU time required to execute task `i`.

The table below shows the value for size `n(2 1/n – 1)` based on the number of tasks. The upper bound for an infinite number of tasks is given by `ln(2)`, or `0.693`, which means that you meet all hard real-time deadlines based on RMS, CPU usage of all time-critical tasks should be less than 70 percent!

Note that you can still have non time-critical tasks in a system and thus use close to 100

Copyright 2015 Micrium Inc.

percent of the CPU’s time. However, using 100 percent of your CPU’s time is not a desirable goal as it does not allow for code changes and added features. As a rule of thumb, always design a system to use less than 60 to 70 percent of the CPU.

RMS says that the highest rate task has the highest priority. In some cases, the highest rate task might not be the most important task. The application should dictate how to assign priorities. However, RMS is an interesting starting point.

Number of Tasks	n(21/n-1)
1	1.00
2	0.828
3	0.779
4	0.756
5	0.743
:	:
:	:
:	:
Infinite	0.693

Table - Allowable CPU usage based on number of tasks

Determining the Size of a Stack

The size of the stack required by the task is application specific. When sizing the stack, however, one must account for the nesting of all the functions called by the task, the number of local variables to be allocated by all functions called by the task, and the stack requirements for all nested interrupt service routines. In addition, the stack must be able to store all CPU registers and possibly Floating-Point Unit (FPU) registers if the processor has a FPU. In addition, as a general rule in embedded systems, avoid writing recursive code.

It is possible to manually figure out the stack space needed by adding all the memory required by all function call nesting (1 pointer each function call for the return address), plus all the memory required by all the arguments passed in those function calls, plus storage for a full CPU context (depends on the CPU), plus another full CPU context for each nested ISRs (if the CPU doesn't have a separate stack to handle ISRs), plus whatever stack space is needed by those ISRs. Adding all this up is a tedious chore and the resulting number is a minimum requirement. Most likely you would not make the stack size that precise in order to account for "surprises." The number arrived at should probably be multiplied by some safety factor, possibly 1.5 to 2.0. This calculation assumes that the exact path of the code is known at all times, which is not always possible. Specifically, when calling a function such as `printf()` or some other library function, it might be difficult or nearly impossible to even guess just how much stack space `printf()` will require. In this case, start with a fairly large stack space and monitor the stack usage at run-time to see just how much stack space is actually used after the application runs for a while.

There are really cool and clever compilers/linkers that provide this information in a link map. For each function, the link map indicates the worst-case stack usage. This feature clearly enables you to better evaluate stack usage for each task. It is still necessary to add the stack space for a **full CPU** context plus, another **full CPU** context for each nested ISR (if the CPU does not have a separate stack to handle ISRs), plus whatever stack space is needed by those ISRs. Again, allow for a safety net and multiply this value by some factor.

Always monitor stack usage at run-time while developing and testing the product as stack overflows occur often and can lead to some curious behaviors. In fact, whenever someone mentions that his or her application behaves "strangely," insufficient stack size is the first thing that comes to mind.

Copyright 2015 Micrium Inc.

Detecting Task Stack Overflows

1) Using an MMU or MPU

Stack overflows are easily detected if the processor has a Memory Management Unit (MMU) or a Memory Protection Unit (MPU). Basically, MMUs and MPUs are special hardware devices integrated alongside the CPU that can be configured to detect when a task attempts to access invalid memory locations, whether code, data, or stack. However, setting up an MMU or MPU is well beyond the scope of this book.

2) Using a CPU with stack overflow detection

Some processors, however, do have simple stack pointer overflow detection registers. When the CPU's stack pointer goes below (or above depending on stack growth) the value set in this register, an exception is generated and the exception handler ensures that the offending code does not do further damage (possibly issue a warning about the faulty code or even terminate it). The `.StkLimitPtr` field in the `OS_TCB` (see Task Control Blocks) is provided for this purpose as shown in the figure below. Note that the position of the stack limit is typically set at a valid location in the task's stack with sufficient room left on the stack to handle the exception itself (assuming the CPU does not have a separate exception stack). In most cases, the position can be fairly close to `&MyTaskStk[0]`.

Copyright 2015 Micrium Inc.

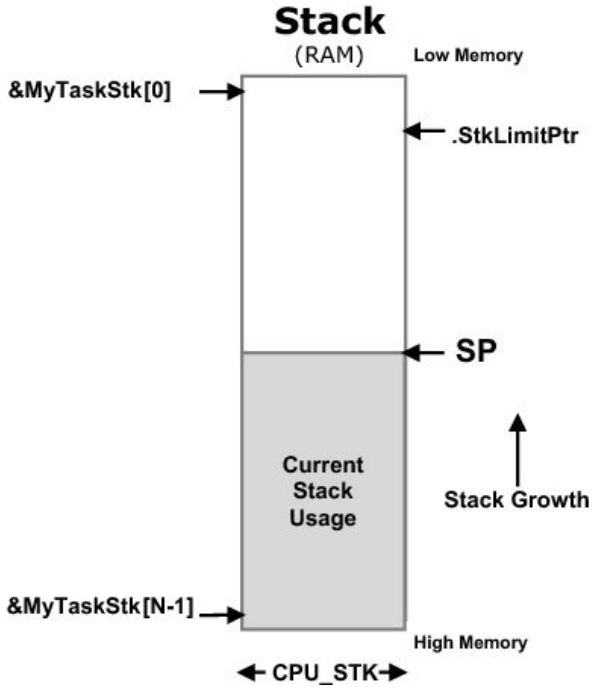


Figure - Hardware detection of stack overflows

As a reminder, the location of the `.StkLimitPtr` is determined by the "`stk_limit`" argument passed to `OSTaskCreate()`, when the task is created as shown below:

```
OS_TCB  MtTaskTCB;
CPU_STK MyTaskStk[1000];


OSTaskCreate(&MyTaskTCB,
             "MyTaskName",
              MyTask,
             &MyTaskArg,
              MyPrio,
             &MyTaskStk[0],   /* Stack base address                                        */
               100,           /* Used to set .StkLimitPtr to trigger exception ...         */
                              /* ... at stack usage > 90%                                  */
              1000,           /* Total stack size (in CPU_STK elements)                    */
              MyTaskQSize,
              MyTaskTimeQuanta,
              (void *)0,
              MY_TASK_OPT,
             &err);
```

Listing - stk_limit, ie .StkLimitPtr, as passed to OSTaskCreate()

Of course, the value of `.StkLimitPtr` used by the CPU's stack overflow detection hardware needs to be changed whenever µC/OS-III performs a context switch. This can be tricky because the value of this register may need to be changed so that it first points to `NULL`, then the

CPU's stack pointer is changed, and finally the value of the stack checking register is set to the value saved in the TCB's `.StkLimitPtr`. Why? Because if the sequence is not followed, the exception could be generated as soon as the stack pointer or the stack overflow detection register is changed. You can avoid this problem by first changing the stack overflow detection register to point to a location that ensures the stack pointer is never invalid (thus the `NULL` as described above). Note that I assumed here that the stack grows from high memory to low memory but the concept works in a similar fashion if the stack grows in the opposite direction.

3) Custom software-based stack overflow detection

Whenever µC/OS-III switches from one task to another, it calls a "hook" function (`OSTaskSwHook()`), which allows the µC/OS-III port programmer to extend the capabilities of the context switch function. So, if the processor doesn't have hardware stack pointer overflow detection, it's still possible to "simulate" this feature by adding code in the context switch hook function and, perform the overflow detection in software. Specifically, before a task is *switched* in, the code should ensure that the stack pointer to load into the CPU does not exceed the "limit" placed in `.StkLimitPtr`. Because the software implementation cannot detect the stack overflow "as soon" as the stack pointer exceeds the value of `.StkLimitPtr`, it is important to position the value of `.StkLimitPtr` in the stack fairly far from `&MyTaskStk[0]`, as shown in the figure below. A software implementation such as this is not as reliable as a hardware-based detection mechanism but still prevents a possible stack overflow. Of course, the `.StkLimitPtr` field would be set using `OSTaskCreate()` as shown above but this time, with a location further away from `&MyTaskStk[0]`.

Copyright 2015 Micrium Inc.

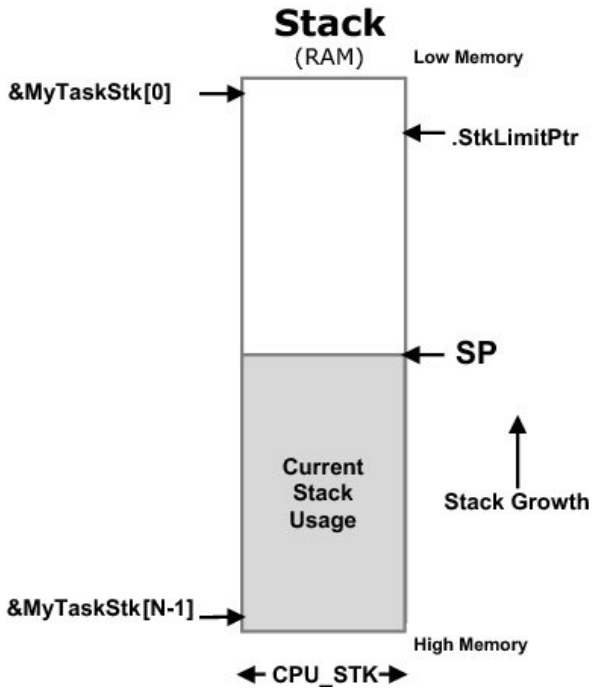


Figure - Software detection of stack overflows, monitoring .StkLimitPtr

4) Redzone stack overflow detection

The Redzone Stack Overflow detection mechanism is built-in within µC/OS-III. This software based approach implements something defined in the previous section and is enabled by setting `OS_CFG_TASK_STK_REDZONE_EN` to `DEF_ENABLED` in `os_cfg.h`. When enabled, µC/OS-III creates a monitored zone at the end of a task's stack. This zone, the Redzone, is filled upon task creation with a special canary like value. The zone is identified in red in the next figure. The figure assumes a stack that grows towards lower addresses. Every time a task needs to be switched out, either at the task level or at the interrupt level, µC/OS-III checks if the Redzone has been overwritten and checks if the stack pointer is still within the limits of the stack. If the zone has been overwritten, or if the stack pointer is out of bounds, µC/OS-III informs the user by calling `OSRedzoneHitHook()`. At that point, it is known that the current task's stack is corrupted. Without an application hook, µC/OS-III simply calls the software based exception. Which stops the execution of the application. See the µC-OS-III API Reference for information regarding `OSRedzoneHitHook()`.

Note that the size of the Redzone is determined by `OS_CFG_TASK_STK_REDZONE_DEPTH`. By default, it is 8 `CPU_STK` elements deep. Hence, the effectively usable stack space is reduced by 8 elements.

Copyright 2015 Micrium Inc.

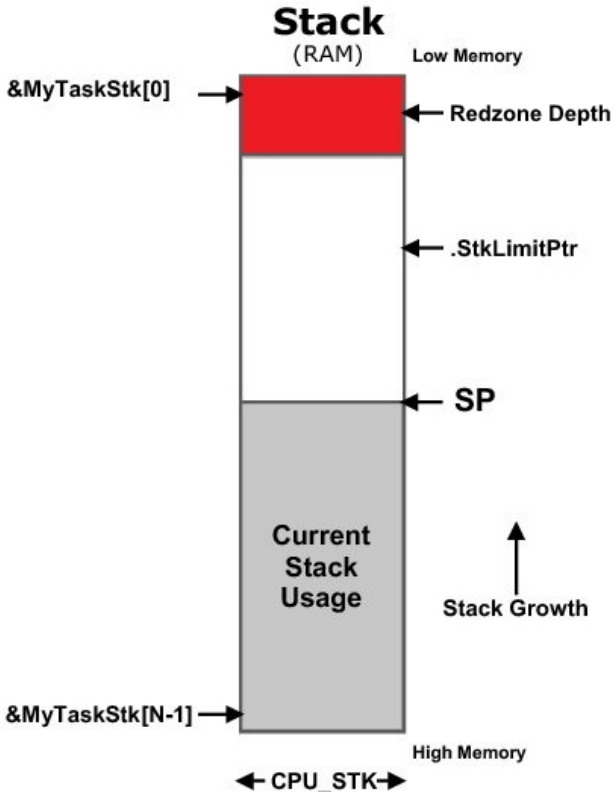


Figure - Redzone Stack Overflow detection

5) Counting the amount of free stack space

Another way to check for stack overflows is to allocate more stack space than is anticipated to be used for the stack, then, monitor and possibly display actual maximum stack usage at run-time. This is fairly easy to do. First, the task stack needs to be cleared (i.e., filled with zeros) when the task is created. Next, a low priority task *walks the stack* of each task created, from the bottom (`&MyTaskStk[0]`) towards the top, counting the number of zero entries. When the task finds a non-zero value, the process is stopped and the usage of the stack can be computed (in number of bytes used or as a percentage). Then, you can adjust the size of the stacks (by recompiling the code) to allocate a more *reasonable* value (either increase or decrease the amount of stack space for each task). For this to be effective, however, you need to run the application long enough for the stack to grow to its highest value. This is illustrated in the figure below. µC/OS-III provides a function that performs this calculation at run-time, `OSTaskStkChk()` and in fact, this function is called by `OS_StatTask()` to compute stack usage for every task created in the application (to be described later).

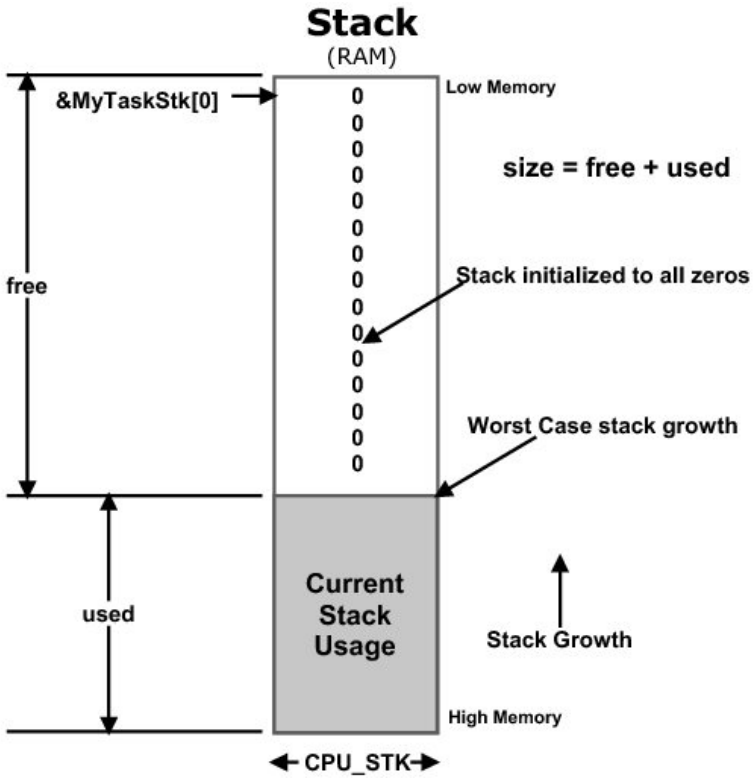


Figure - Software detection of stack overflows, walking the stack

Task Management Services

µC/OS-III provides a number of task-related services to call from the application. These services are found in `os_task.c` and they all start with `OSTask???()`. The type of service they perform groups task-related services:

Group	Functions

General	`OSTaskCreate()`
	`OSTaskDel()`
	`OSTaskChangePrio()`
	`OSTaskRegSet()`
	`OSTaskRegGet()`
	`OSTaskSuspend()`
	`OSTaskResume()`
	`OSTaskTimeQuantaSet()`

Signaling a Task
(See Synchronization)
	`OSTaskSemPend()`
	`OSTaskSemPost()`
	`OSTaskSemPendAbort()`

Sending Messages to a Task
(See Message Passing)
	`OSTaskQPend()`
	`OSTaskQPost()`
	`OSTaskQPendAbort()`
	`OSTaskQFlush()`

Table - Task Management Services

A complete description of all µC/OS-III task related services is provided in µC/OS-III API Reference.

Copyright 2015 Micrium Inc.

Task Management Internals

Task States

From a µC/OS-III user point of view, a task can be in any one of five states as shown in the figure below. Internally, µC/OS-III does not need to keep track of the dormant state and the other states are tracked slightly differently. The actual µC/OS-III states will be discussed after a discussion on task states from the user’s point of view. The figure below also shows which µC/OS-III functions are used to move from one state to another. The diagram is actually simplified as state transitions are a bit more complicated than this.

Copyright 2015 Micrium Inc.

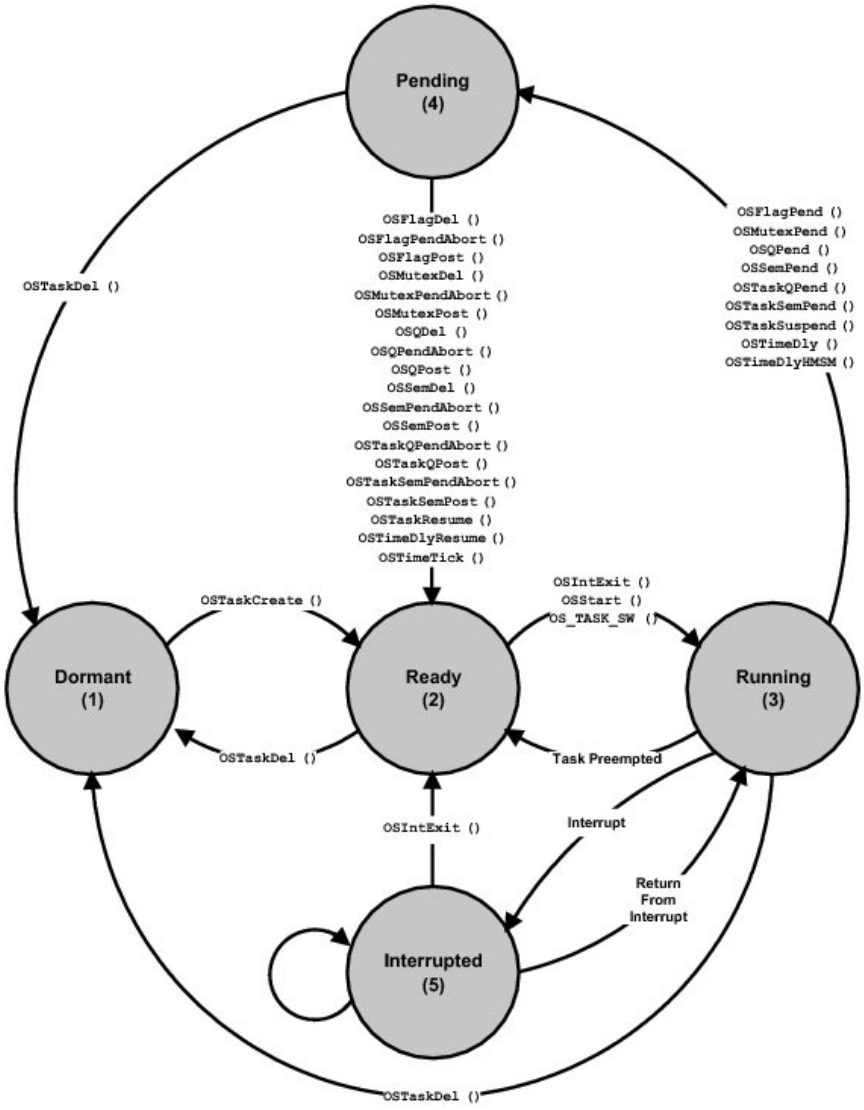


Figure - Five basic states of a task

(1) The *Dormant* state corresponds to a task that resides in memory but has not been made available to µC/OS-III.

A task is made available to µC/OS-III by calling a function to create the task, `OSTaskCreate()`. The task code actually resides in code space but µC/OS-III needs to be informed about it.

When it is no longer necessary for µC/OS-III to manage a task, your code can call the

task delete function, `OSTaskDel()`. `OSTaskDel()` does not actually delete the code of the task, it is simply not eligible to access the CPU.

(2) A task is in the *Ready* state when it is ready-to-run. There can be any number of tasks ready and µC/OS-III keeps track of all ready tasks in a `ready list` (discussed later). This list is sorted by priority.

(3) The most important ready-to-run task is placed in the *Running* state. On a single CPU, only one task can be running at any given time.

The task selected to run on the CPU is *switched in* by µC/OS-III when the application code calls `OSStart()`, or when µC/OS-III calls either `OSIntExit()` or `OS_TASK_SW()`.

As previously discussed, tasks must wait for an event to occur. A task waits for an event by calling one of the functions that brings the task to the pending state if the event has not occurred.

(4) Tasks in the Pending state are placed in a special list called a *pend-list* (or wait list) associated with the event the task is waiting for. When waiting for the event to occur, the task does not consume CPU time. When the event occurs, the task is placed back into the ready list and µC/OS-III decides whether the newly readied task is the most important ready-to-run task. If this is the case, the currently running task will be preempted (placed back in the ready list) and the newly readied task is given control of the CPU. In other words, the newly readied task will run immediately if it is the most important task.

Note that the `OSTaskSuspend()` function unconditionally blocks a task and this task will not actually wait for an event to occur but in fact, waits until another task calls `OSTaskResume()` to make the task ready-to-run.

(5) Assuming that CPU interrupts are enabled, an interrupting device will suspend execution of a task and execute an Interrupt Service Routine (ISR). ISRs are typically events that tasks wait for. Generally speaking, an ISR should simply notify a task that an event occurred and let the task process the event. ISRs should be as short as possible and most of the work of handling the interrupting devices should be done at the task level where it can be managed by µC/OS-III. ISRs are only allowed to make “Post” calls (i.e., `OSFlagPost()`, `OSQPost()`, `OSSemPost()`, `OSTaskQPost()` and `OSTaskSemPost()`). The only post call not allowed to be made from an ISR is `OSMutexPost()` since mutexes, as will be addressed later, are assumed to be services that are only accessible at the task level.

Copyright 2015 Micrium Inc.

As the state diagram indicates, an interrupt can interrupt another interrupt. This is called `interrupt nesting` and most processors allow this. However, interrupt nesting easily leads to stack overflow if not managed properly.

Internally, µC/OS-III keeps track of task states using the state machine shown in the figure below. The task state is actually maintained in a variable that is part of a data structure associated with each task, the task's TCB. The task state diagram was referenced throughout the design of µC/OS-III when implementing most of µC/OS-III's services. The number in parentheses is the state number of the task and thus, a task can be in any one of eight (8) states (see `os.h, OS_TASK_STATE_???`).

Note that the diagram does not keep track of a dormant task, as a dormant task is not known to µC/OS-III. Also, interrupts and interrupt nesting is tracked differently as will be explained further in the text.

This state diagram should be quite useful to understand how to use several functions and their impact on the state of tasks. In fact, I'd highly recommend that the reader bookmark the page of the diagram.

Copyright 2015 Micrium Inc.

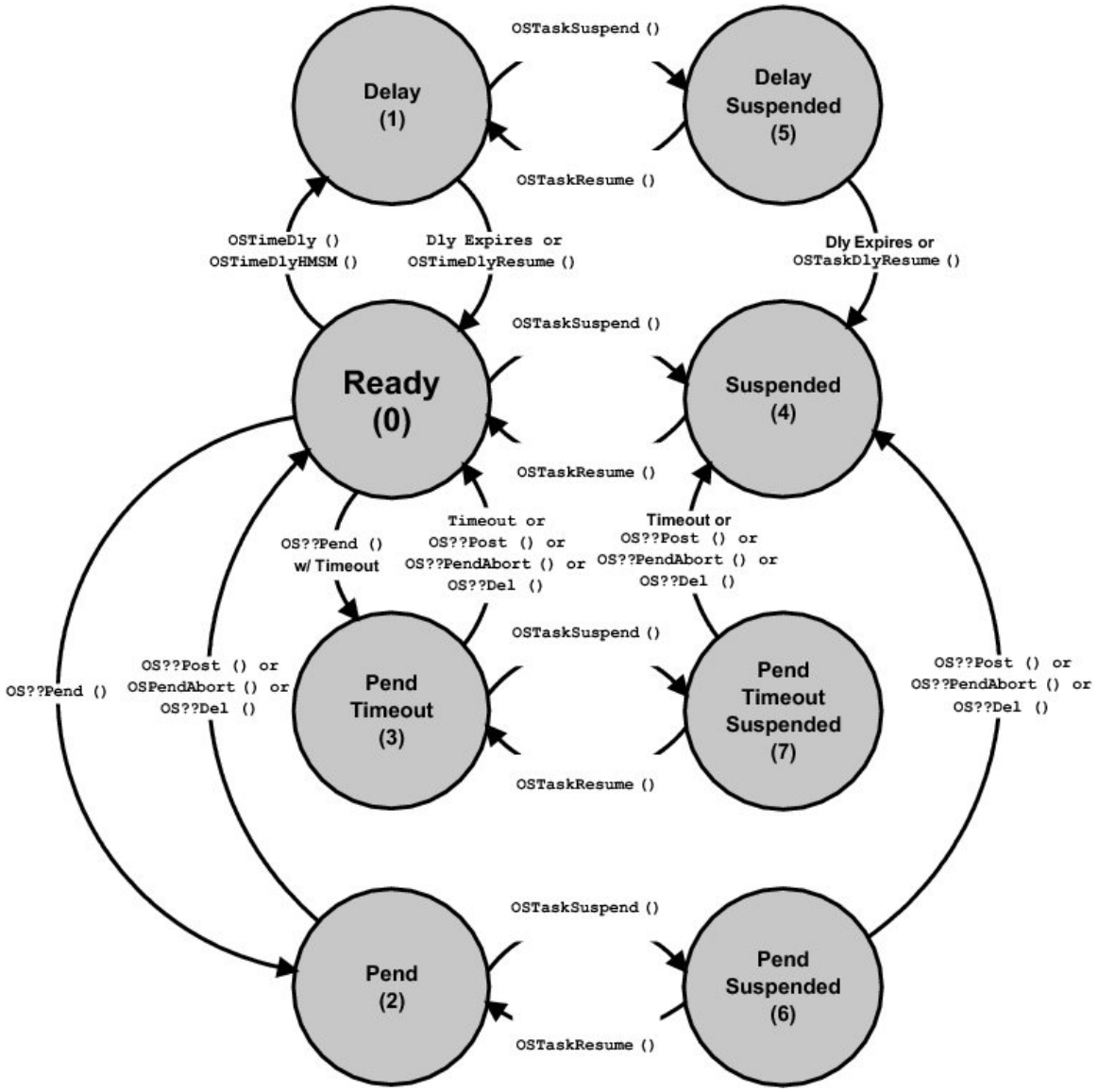


Figure - µC/OS-III's internal task state machine

(1) A task is in State 0 when a task is ready-to-run. Every task "wants" to be ready-to-run as that is the only way it gets to perform their duties.

(2) A task can decide to wait for time to expire by calling either `OSTimeDly()` or `OSTimeDlyHMSM()`. When the time expires or the delay is cancelled (by calling `OSTimeDlyResume()`), the task returns to the ready state.

Copyright 2015 Micrium Inc.

(3) A task can wait for an event to occur by calling one of the pend (i.e., wait) functions (`OSFlagPend()`, `OSMutexPend()`, `OSQPend()`, `OSSemPend()`, `OSTaskQPend()`, or `OSTaskSemPend()`), and specify to wait forever for the event to occur. The pend terminates when the event occurs (i.e., a task or an ISR performs a “post”), the awaited object is deleted or, another task decides to abort the pend.

(4) A task can wait for an event to occur as indicated, but specify that it is willing to wait a certain amount of time for the event to occur. If the event is not posted within that time, the task is readied, then the task is notified that a timeout occurred. Again, the pend terminates when the event occurs (i.e., a task or an ISR performs a “post”), the object awaited is deleted or, another task decides to abort the pend.

(5) A task can suspend itself or another task by calling `OSTaskSuspend()`. The only way the task is allowed to resume execution is by calling `OSTaskResume()`. Suspending a task means that a task will not be able to run on the CPU until it is resumed. If a task suspends itself then it must be resumed by another task.

(6) A delayed task can also be suspended by another task. In this case, the effect is additive. In other words, the delay must complete (or be resumed by `OSTimeDlyResume()`) and the suspension must be removed (by another task which would call `OSTaskResume()`) in order for the task to be able to run.

(7) A task waiting on an event to occur may be suspended by another task. Again, the effect is additive. The event must occur and the suspension removed (by another task) in order for the task to be able to run. Of course, if the object that the task is pending on is deleted or, the pend is aborted by another task, then one of the above two condition is removed. The suspension, however, must be explicitly removed.

(8) A task can wait for an event, but only for a certain amount of time, and the task could also be suspended by another task. As one might expect, the suspension must be removed by another task (or the same task that suspended it in the first place), and the event needs to either occur or timeout while waiting for the event.

Copyright 2015 Micrium Inc.

Task Control Blocks

A task control block (TCB) is a data structure used by kernels to maintain information about a task. Each task requires its own TCB and, for µC/OS-III, the user assigns the TCB in user memory space (RAM). The address of the task's TCB is provided to µC/OS-III when calling task-related services (i.e., `OSTask???()` functions). The task control block data structure is declared in `os.h` as shown in the listing below. Note that the fields are actually commented in `os.h`, and some of the fields are conditionally compiled based on whether or not certain features are desired. Both are not shown here for clarity.

Also, it is important to note that even when the user understands what the different fields of the `OS_TCB` do, the application code must never directly access these (especially change them). In other words, `OS_TCB` fields must only be accessed by µC/OS-III and not the code.

Copyright 2015 Micrium Inc.

```
struct os_tcb {
    CPU_STK             *StkPtr;
    void                *ExtPtr;
    CPU_STK             *StkLimitPtr;
    OS_TCB              *NextPtr;
    OS_TCB              *PrevPtr;
    OS_TCB              *TickNextPtr;
    OS_TCB              *TickPrevPtr;
    OS_TICK_LIST        *TickListPtr;
    OS_CHAR             *NamePtr;
    CPU_STK             *StkBasePtr;
    OS_TLS               TLS_Tbl[OS_CFG_TLS_TBL_SIZE]
    OS_TASK_PTR          TaskEntryAddr;
    void                *TaskEntryArg;
    OS_PEND_DATA        *PendDataTblPtr;
    OS_STATE             PendOn;
    OS_STATUS            PendStatus;
    OS_STATE             TaskState;
    OS_PRIO              Prio;
    OS_PRIO              BasePrio;
    OS_MUTEX            *MutexGrpHeadPtr;
    CPU_STK_SIZE         StkSize;
    OS_OPT               Opt;
    OS_OBJ_QTY           PendDataEntries;
    CPU_TS               TS;
    OS_SEM_CTR           SemCtr;
    OS_TICK              TickRemain;
    OS_TICK              TickCtrPrev;
    OS_TICK              TimeQuanta;
    OS_TICK              TimeQuantaCtr;
    void                *MsgPtr;
    OS_MSG_SIZE          MsgSize;
    OS_MSG_Q             MsgQ;
    CPU_TS               MsgQPendTime;
    CPU_TS               MsgQPendTimeMax;
    OS_REG               RegTbl[OS_TASK_REG_TBL_SIZE];
    OS_FLAGS             FlagsPend;
    OS_FLAGS             FlagsRdy;
    OS_OPT               FlagsOpt;
    OS_NESTING_CTR       SuspendCtr;
    OS_CPU_USAGE         CPUUsage;
    OS_CPU_USAGE         CPUUsageMax;
    OS_CTX_SW_CTR        CtxSwCtr;
    CPU_TS               CyclesDelta;
    CPU_TS               CyclesStart;
    OS_CYCLES            CyclesTotal;
    OS_CYCLES            CyclesTotalPrev;
    CPU_TS               SemPendTime;
    CPU_TS               SemPendTimeMax;
    CPU_STK_SIZE         StkUsed;
    CPU_STK_SIZE         StkFree;
    CPU_TS               IntDisTimeMax;
    CPU                  SchedLockTimeMax;
    OS_TCB               DbgNextPtr;
    OS_TCB               DbgPrevPtr;
    CPU_CHAR             DbgNamePtr;
};
```

Listing - OS_TCB Data Structure

`.StkPtr`

This field contains a pointer to the current top-of-stack for the task. µC/OS-III allows each task to have its own stack and each stack can be any size. `.StkPtr` should be the only field in the `OS_TCB` data structure accessed from assembly language code (for the context-switching code). This field is therefore placed as the first entry in the structure making access easier from assembly language code (it will be at offset zero in the data structure).

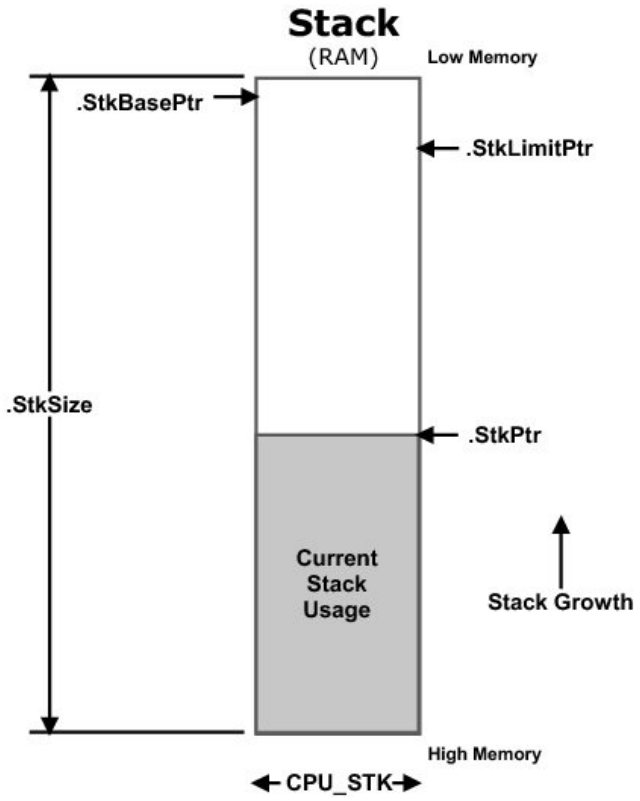
`.ExtPtr`

This field contains a pointer to a user-definable data area used to extend the TCB as needed. This pointer is provided as an argument passed in `OSTaskCreate()`. This pointer is easily accessible from assembly language since it always follows the `.StkPtr`. `.ExtPtr` can be used to add storage for saving the context of a FPU (Floating-Point Unit) if the processor you are using has a FPU.

`.StkLimitPtr`

The field contains a pointer to a location in the task's stack to set a *watermark* limit for stack growth and is determined from the value of the "`stk_limit`" argument passed to `OSTaskCreate()`. Some processors have special registers that automatically check the value of the stack pointer at run-time to ensure that the stack does not overflow. `.StkLimitPtr` may be used to set this register during a context switch. Alternatively, if the processor does not have such a register, this can be "simulated" in software. However, this is not as reliable as a hardware solution. If this feature is not used then you can set the value of "`stk_limit`" can be set to `0` when calling `OSTaskCreate()`. See also Detecting Task Stack Overflows).

Copyright 2015 Micrium Inc.



`.NextPtr and .PrevPtr`

These pointers are used to doubly link `OS_TCBs` in the ready list. A doubly linked list allows `OS_TCBs` to be quickly inserted and removed from the list.

`.TickNextPtr and .TickPrevPtr`

These pointers are used to doubly link `OS_TCBs` in the list of tasks waiting for time to expire or to timeout from pend calls. Again, a doubly linked list allows `OS_TCBs` to be quickly inserted and removed from the list.

`.TickListPtr`

This pointer points to one of two tick lists: either the list of tasks delayed waiting for time to expire (`OSTickListDly`) or tasks pending on an object with a timeout (`OSTickListTimeout`). This field is used to easily remove the `OS_TCB` from the list.

`.NamePtr`

Copyright 2015 Micrium Inc.

This pointer allows a name (an ASCII string) to be assigned to each task. Having a name is useful when debugging, since it is user friendly compared to displaying the address of the `OS_TCB`. Storage for the ASCII string is assumed to be in user space, either in code memory (ASCII string declared as a `const`) or in RAM.

`.StkBasePtr`

This field points to the base address of the task's stack. The stack base is typically the lowest address in memory where the stack for the task resides. A task stack is declared as follows:

```
CPU_STK MyTaskStk[???];
```

`CPU_STK` is the data type you must use to declare task stacks and `???` is the size of the stack associated with the task. The base address is always `&MyTaskStk[0]`.

`.TLS_Tbl[]`

This field typically contains an array of pointers and is used by compilers for thread safety as described in Chapter 20, "Thread Safety of the Compiler's Run-Time Library").

`.TaskEntryAddr`

This field contains the entry address of the task. As previously mentioned, a task is declared as shown below and this field contains the address of `MyTask`.

```
void MyTask (void *p_arg);
```

`.TaskEntryArg`

This field contains the value of the argument that is passed to the task when the task starts. As previously mentioned, a task is declared as shown below and this field contains the value of `p_arg`.

```
void MyTask (void *p_arg);
```

`.PendDataTblPtr`

µC/OS-III allows the task to pend on any number of semaphores or message queues simultaneously. This pointer points to a table containing information about the pended objects. This is described in Chapter 10, “Pend Lists”.

`.PendOn`

This field indicates what the task is pending on and contains one of the following values declared in `os.h`:

```
OS_TASK_PEND_ON_NOTHING
OS_TASK_PEND_ON_FLAG
OS_TASK_PEND_ON_TASK_Q
OS_TASK_PEND_ON_MULTI
OS_TASK_PEND_ON_MUTEX
OS_TASK_PEND_ON_Q
OS_TASK_PEND_ON_SEM
OS_TASK_PEND_ON_TASK_SEM
```

`.PendStatus`

This field indicates the outcome of a pend and contains one of the values declared in `os.h`:

```
OS_STATUS_PEND_OK
OS_STATUS_PEND_ABORT
OS_STATUS_PEND_DEL
OS_STATUS_PEND_TIMEOUT
```

`.TaskState`

This field indicates the current state of a task and contains one of the eight (8) task states that a task can be in. These states are declared in `os.h`:

```
OS_TASK_STATE_RDY
OS_TASK_STATE_DLY
OS_TASK_STATE_PEND
OS_TASK_STATE_PEND_TIMEOUT
OS_TASK_STATE_SUSPENDED
OS_TASK_STATE_DLY_SUSPENDED
```

Copyright 2015 Micrium Inc.

```
OS_TASK_STATE_PEND_SUSPENDED
OS_TASK_STATE_PEND_TIMEOUT_SUSPENDED
```

`.Prio`

This field contains the current priority of a task. `.Prio` is a value between 0 and `OS_CFG_PRIO_MAX-1`. In fact, the idle task is the only task at priority `OS_CFG_PRIO_MAX-1`.

`.BasePrio`

This field contains the base priority of a task. `.BasePrio` is a value between 0 and `OS_CFG_PRIO_MAX-1`. This field is used to correctly propagate mutex owner priorities to prevent priority inversion.

`.MutexGrpHeadPtr`

This field is a pointer to a list of mutexes that this task owns.

`.StkSize`

This field contains the size (in number of `CPU_STK` elements) of the stack associated with the task. Recall that a task stack is declared as follows:

```
CPU_STK MyTaskStk[???];
```

`.StkSize` is the number of elements in the above array.

`.Opt`

This field saves the “options” passed to `OSTaskCreate()` when the task is created as shown below. Note that task options are additive.

```
OS_OPT_TASK_NONE
OS_OPT_TASK_STK_CHK
OS_OPT_TASK_STK_CLR
OS_OPT_TASK_SAVE_FP
```

`.PendDataTblEntries`

This field works with the `.PendDataTblPtr` and indicates the number of objects a task is pending on at the same time.

`.TS`

This field is used to store a “time stamp” of when an event that the task was waiting on occurred. When the task resumes execution, this time stamp is returned to the caller.

`.SemCtr`

This field contains a semaphore counter associated with the task. Each task has its own built-in semaphore. An ISR or another task can signal a task using this semaphore. `.SemCtr` is therefore used to keep track of how many times the task is signaled. `.SemCtr` is used by `OSTaskSem???()` services.

`.TickRemain`

This field contains the number of ticks remaining before the task is removed from either the delayed or timeout list. As of V3.04.00, the value of this field is relative (it used to be absolute) since these lists are now implemented as delta-lists. In other words, this field does not contain absolute time remaining until the delay or timeout expires.

`.TickCtrPrev`

This field is used to keep track of the previous value of `OSTickCtr` in order to be able to implement the periodic mode for `OSTimeDly()`.

`.TimeQuanta and .TimeQuantaCtr`

These fields are used for time slicing. When multiple tasks are ready-to-run at the same priority, `.TimeQuanta` determines how much time (in ticks) the task will execute until it is preempted by µC/OS-III so that the next task at the same priority gets a chance to execute. `.TimeQuantaCtr` keeps track of the remaining number of ticks for this to happen and is loaded with `.TimeQuanta` at the beginning of the task’s time slice.

`.MsgPtr`

When a message is sent to a task, this field contains the message received. This field only

Copyright 2015 Micrium Inc.

exists in a TCB if message queue services (OS_CFG_Q_EN is set to DEF_ENABLED in os_cfg.h), or task message queue services, are enabled (OS_CFG_TASK_Q_EN is set to DEF_ENABLED in os_cfg.h) at compile time.

.MsgSize

When a message is sent to a task, this field contains the size (in number of bytes) of the message received. This field only exists in a TCB if message queue services (OS_CFG_Q_EN is set to DEF_ENABLED in os_cfg.h), or task message queue services, (OS_CFG_TASK_Q_EN is set to DEF_ENABLED in os_cfg.h) are enabled at compile time.

.MsgQ

µC/OS-III allows tasks or ISRs to send messages directly to tasks. Because of this, a message queue is actually built into each TCB. This field only exists in a TCB if task message queue services are enabled at compile time (OS_CFG_TASK_Q_EN is set to DEF_ENABLED in os_cfg.h). .MsgQ is used by the OSTaskQ???() services.

.MsgQPendTime

This field contains the amount of time it took for a message to arrive. When OSTaskQPost() is called, the current time stamp is read and stored in the message. When OSTaskQPend() returns, the current time stamp is read again and the difference between the two times is stored in this variable. A debugger or µC/Probe can be used to indicate the time taken for a message to arrive by displaying this field.

This field is only available if setting OS_CFG_TASK_PROFILE_EN to DEF_ENABLED in os_cfg.h.

.MsgQPendTimeMax

This field contains the maximum amount of time it takes for a message to arrive. It is a peak detector of the value of .MsgQPendTime. The peak can be reset by calling OSStatReset().

This field is only available if setting OS_CFG_TASK_PROFILE_EN to DEF_ENABLED in os_cfg.h.

.RegTbl[]

Copyright 2015 Micrium Inc.

This field contains a table of “registers” that are task-specific. These registers are different than CPU registers. Task registers allow for the storage of such task-specific information as task ID, “errno” common in some software components, and more. Task registers may also store task-related data that needs to be associated with the task at run time. Note that the data type for elements of this array is OS_REG, which can be declared at compile time to be nearly anything. However, all registers must be of this data type. This field only exists in a TCB if task registers are enabled at compile time (OS_CFG_TASK_REG_TBL_SIZE is greater than 0 in os_cfg.h).

.FlagsPend

When a task pends on event flags, this field contains the event flags (i.e., bits) that the task is pending on. This field only exists in a TCB if event flags services are enabled at compile time (OS_CFG_FLAG_EN is set to DEF_ENABLED in os_cfg.h).

.FlagsRdy

This field contains the event flags that were posted and that the task was waiting on. In other words, it allows a task to know which event flags made the task ready-to-run. This field only exists in a TCB if event flags services are enabled at compile time (OS_CFG_FLAG_EN is set to DEF_ENABLED in os_cfg.h).

.FlagsOpt

When a task pends on event flags, this field contains the type of pend (pend on any event flag bit specified in .FlagsPend or all event flag bits specified in .FlagsPend). This field only exists in a TCB if event flags services are enabled at compile time (OS_CFG_FLAG_EN is set to DEF_ENABLED in os_cfg.h). There can be up to eight main values as shown below plus add-on options. Possible values are:

```
OS_OPT_PEND_FLAG_CLR_ALL
OS_OPT_PEND_FLAG_CLR_ANY
OS_OPT_PEND_FLAG_CLR_AND
OS_OPT_PEND_FLAG_CLR_OR
OS_OPT_PEND_FLAG_SET_ALL
OS_OPT_PEND_FLAG_SET_ANY
OS_OPT_PEND_FLAG_SET_AND
OS_OPT_PEND_FLAG_SET_OR
```

Copyright 2015 Micrium Inc.

You can also ‘add’ OS_OPT_PEND_FLAG_CONSUME and either OS_OPT_PEND_BLOCKING or OS_OPT_PEND_NON_BLOCKING to the above options.

.SuspendCtr

This field is used by OSTaskSuspend() and OSTaskResume() to keep track of how many times a task is suspended. Task suspension can be nested. When .SuspendCtr is 0, all suspensions are removed. This field only exists in a TCB if task suspension is enabled at compile time (OS_CFG_TASK_SUSPEND_EN is set to DEF_ENABLED in os_cfg.h).

.CPUUsage

This field is computed by OS_StatTask() if OS_CFG_TASK_PROFILE_EN is set to DEF_ENABLED in os_cfg.h. .CPUUsage contains the CPU usage of a task in percent (0 to 100%). As of version V3.03.00, .CPUUsage is multiplied by 100. In other words, 10000 represents 100.00%.

.CPUUsageMax

This field contains the maximum (i.e., peak detector) CPU usage in percentage for the task and is only available if setting OS_CFG_TASK_PROFILE_EN to DEF_ENABLED in os_cfg.h.

This field is computed by OS_StatTask(). As of version V3.03.00, .CPUUsageMax represents percentage multiplied by 100. In other words, 10000 represents 100.00%.

The peak can be reset by calling OSStatReset().

.CtxSwCtr

This field keeps track of how often the task has executed (not how long it has executed). This field is generally used by debuggers or run-time monitors to see if a task is executing (the value of this field would be non-zero and would be incrementing). The field is enabled at compile time when OS_CFG_TASK_PROFILE_EN is set to DEF_ENABLED in os_cfg.h.

.CyclesDelta

.CyclesDelta is computed during a context switch and contains the value of the current time stamp (obtained by calling OS_TS_GET()) minus the value of .CyclesStart. This field

is generally used by debuggers or a run-time monitor to see how long a task had control of the CPU until it got switched out. The field is enabled at compile time when OS_CFG_TASK_PROFILE_EN is set to DEF_ENABLED in os_cfg.h.

.CyclesStart

This field is used to measure how long a task had control of the CPU. .CyclesStart is updated when µC/OS-III performs a context switch. .CyclesStart contains the value of the current time stamp (it calls OS_TS_GET()) when a task is switched-in. The field is enabled at compile time when OS_CFG_TASK_PROFILE_EN is set to DEF_ENABLED in os_cfg.h.

.CyclesTotal

This field accumulates the value of .CyclesDelta, so it contains the total execution time of a task during a set period of time. .CyclesTotal is used by OS_StatTask() to determine CPU usage on a per-task basis. This is typically a 32-bit value because of the accumulation of cycles over time. On the other hand, using a 64-bit value ensures that we can accumulate CPU cycles for almost 600 years even if the CPU is running at 1 GHz! Of course, it's assumed that the compiler supports 64-bit data types. The field is enabled at compile time when OS_CFG_TASK_PROFILE_EN is set to DEF_ENABLED in os_cfg.h.. .CyclesTotal is used by OS_StatTask() to determine CPU usage on a per-task basis.

.SemPendTime

This field contains the amount of time taken for the semaphore to be signaled. When OSTaskSemPost() is called, the current time stamp is read and stored in the OS_TCB (see .TS). When OSTaskSemPend() returns, the current time stamp is read again and the difference between the two times is stored in this variable. This field can be displayed by a debugger or µC/Probe to indicate how much time it took for the task to be signaled.

This field is only available when setting OS_CFG_TASK_PROFILE_EN to DEF_ENABLED in os_cfg.h.

.SemPendTimeMax

This field contains the maximum amount of time it took for the task to be signaled. It is a peak detector of the value of .SemPendTime. The peak can be reset by calling OSStatReset().

Copyright 2015 Micrium Inc.

This field is only available if setting OS_CFG_TASK_PROFILE_EN to DEF_ENABLED in os_cfg.h.

.StkUsed and .StkFree

µC/OS-III is able to compute (at run time) the amount of stack space a task actually uses and how much stack space remains. This is accomplished by a function called OSTaskStkChk(). Stack usage computation assumes that the task's stack is "cleared" when the task is created. In other words, when calling OSTaskCreate(), it is expected that the following options be specified: OS_TASK_OPT_STK_CLR and OS_TASK_OPT_STK_CHK. OSTaskCreate() will then clear all the RAM used for the task's stack.

µC/OS-III provides an internal task called OS_StatTask() that checks the stack of each of the tasks at run-time. OS_StatTask() typically runs at a low priority so that it does not interfere with the application code. OS_StatTask() saves the value computed for each task in the TCB of each task in these fields, which represents the maximum number of stack bytes used and the amount of stack space still unused by the task. These fields only exist in a TCB if the statistic task is enabled at compile time (OS_CFG_STAT_TASK_STK_CHK_EN is set to DEF_ENABLED in os_cfg.h).

.IntDisTimeMax

This field keeps track of the maximum interrupt disable time of the task. The field is updated only if µC/CPU supports interrupt disable time measurements. This field is available only if setting OS_CFG_TASK_PROFILE_EN to DEF_ENABLED in os_cfg.h and µC/CPU's CPU_CFG_INT_DIS_MEAS_EN is defined in cpu_cfg.h.

.SchedLockTimeMax

The field keeps track of the maximum scheduler lock time of the task. In other words, the maximum amount of time the task locks the scheduler.

This field is available only if you set OS_CFG_TASK_PROFILE_EN to DEF_ENABLED and OS_CFG_SCHED_LOCK_TIME_MEAS_EN is set to DEF_ENABLED in os_cfg.h.

.DbgNextPtr

This field contains a pointer to the next OS_TCB in a doubly linked list of OS_TCBs. OS_TCBs

are placed in this list by OSTaskCreate(). This field is only present if OS_CFG_DBG_EN is set to DEF_ENABLED in os_cfg.h.

.DbgPrevPtr

This field contains a pointer to the previous OS_TCB in a doubly linked list of OS_TCBs. OS_TCBs are placed in this list by OSTaskCreate(). This field is only present if OS_CFG_DBG_EN is set to DEF_ENABLED in os_cfg.h.

.DbgNamePtr

This field contains a pointer to the name of the object that the task is pending on when the task is pending on either an event flag group, a semaphore, a mutual exclusion semaphore or a message queue. This information is quite useful during debugging and thus, this field is only present if OS_CFG_DBG_EN is set to DEF_ENABLED in os_cfg.h.

Copyright 2015 Micrium Inc.

Internal Tasks

During initialization, µC/OS-III creates a minimum of two (2) internal tasks (`OS_IdleTask()` and `OS_TickTask()`) and, three (3) optional tasks (`OS_StatTask()`, `OS_TmrTask()` and `OS_IntQTask()`). The optional tasks are created based on the value of compile-time `#defines` found in `os_cfg.h`:

- `OS_CFG_STAT_TASK_EN` enables `OS_StatTask()`
- `OS_CFG_TMR_EN` enables `OS_TmrTask()`
- `OS_CFG_ISR_POST_DEFERRED_EN` enables `OS_IntQTask()`

Advanced applications may not require the Tick task or the Idle task. To disable a task, set its configuration option to `DEF_DISABLED` in `os_cfg.h`. The following configuration parameters affect the Tick and Idle tasks, respectively:

- `OS_CFG_TASK_TICK_EN`
- `OS_CFG_TASK_IDLE_EN`

Copyright 2015 Micrium Inc.

The Idle Task

If enabled, `OS_IdleTask()` is the very first task created by µC/OS-III. The priority of the idle task is always set to `OS_CFG_PRIO_MAX-1`. In fact, `OS_IdleTask()` is the only task that is ever allowed to be at this priority and, as a safeguard, when other tasks are created, `OSTaskCreate()` ensures that there are no other tasks created at the same priority as the idle task. The idle task runs whenever there are no other tasks that are ready-to-run.

If RAM space is tight, the idle task can be removed. This allows the user to save some data space since there is no need to have a stack for the idle task. However, even if there is not an actual task running at priority `OS_CFG_PRIO_MAX-1`, the user cannot create a task at that priority level. The level is still used to determine when to idle the CPU.

The important portions of the code for the idle task are shown below (refer to `os_core.c` for the complete code). These portions are still executed even if the idle task is disabled in `os_cfg.h`.

```
void  OS_IdleTask (void *p_arg)
          {
              while (DEF_ON) {                    (1)
                  CPU_CRITICAL_ENTER();
                  OSIdleTaskCtr++;                (2)
                  OSStatTaskCtr++;
                  CPU_CRITICAL_EXIT();
                  OSIdleTaskHook();               (3)
              }
          }
```

Listing - Idle Task

(1) The idle task is a “true” infinite loop that never calls functions to “wait for an event”. This is because, on most processors, when there is “nothing to do,” the processor still executes instructions. When µC/OS-III determines that there is no other higher-priority task to run, µC/OS-III “parks” the CPU in the idle task. However, instead of having an empty “infinite loop” doing nothing, µC/OS-III uses this “idle” time to do something useful.

(2) Two counters are incremented whenever the idle task runs.

`OSIdleTaskCtr` is typically defined as a 32-bit unsigned integer (see `os.h`). `OSIdleTaskCtr` is reset once when µC/OS-III is initialized. `OSIdleTaskCtr` is used to indicate “activity”

in the idle task. In other words, if your code monitors and displays `OSIdleTaskCtr`, you should expect to see a value between `0x00000000` and `0xFFFFFFFF`. The rate at which `OSIdleTaskCtr` increments depend on how busy the CPU is at running the application code. The faster the increment, the less work the CPU has to do in application tasks.

`OSStatTaskCtr` is also typically defined as a 32-bit unsigned integer (see `os.h`) and is used by the statistic task (described later) to get a sense of CPU utilization at run time.

(3) Every time through the loop, `OS_IdleTask()` calls `OSIdleTaskHook()`, which is a function that is declared in the µC/OS-III port for the processor used. `OSIdleTaskHook()` allows the implementer of the µC/OS-III port to perform additional processing during idle time. It is very important for this code to not make calls that would cause the idle task to "wait for an event". This is generally not a problem as most programmers developing µC/OS-III ports know to follow this simple rule.

`OSIdleTaskHook()` may be used to place the CPU in low-power mode for battery-powered applications and thus avoid wasting energy. However, doing this means that `OSStatTaskCtr` cannot be used to measure CPU utilization (described later).

```
void  OSIdleTaskHook (void)
          {
              /* Place the CPU in low power mode */
          }
```

Typically, most processors exit low-power mode when an interrupt occurs. Depending on the processor, however, the Interrupt Service Routine (ISR) may have to write to "special" registers to return the CPU to its full or desired speed. If the ISR wakes up a high-priority task (every task is higher in priority than the idle task) then the ISR will not immediately return to the interrupted idle task, but instead switch to the higher-priority task. When the higher-priority task completes its work and waits for its event to occur, µC/OS-III causes a context switch to return to `OSIdleTaskHook()` just "after" the instruction that caused the CPU to enter low-power mode. In turn, `OSIdleTaskHook()` returns to `OS_IdleTask()` and causes another iteration through the "infinite loop" which places the CPU back in the low power state.

Copyright 2015 Micrium Inc.

The Tick Task

Periodic Tick

Nearly every RTOS requires a periodic time source called a *Clock Tick* or *System Tick* to keep track of time delays and timeouts. µC/OS-III's clock tick handling is encapsulated in the file `os_tick.c`.

`OS_TickTask()` is a task created by µC/OS-III and its priority is configurable by the user through µC/OS-III's configuration file `os_cfg_app.h` (see `OS_CFG_TICK_TASK_PRIO`). Typically `OS_TickTask()` is set to a relatively high priority but should be slightly lower in priority than your most important tasks.

`OS_TickTask()` is used by µC/OS-III to keep track of tasks waiting for time to expire or, for tasks that are pending on kernel objects with a timeout. `OS_TickTask()` is a periodic task and it waits for signals from the tick ISR (described in Interrupt Management) as shown in the figure below.

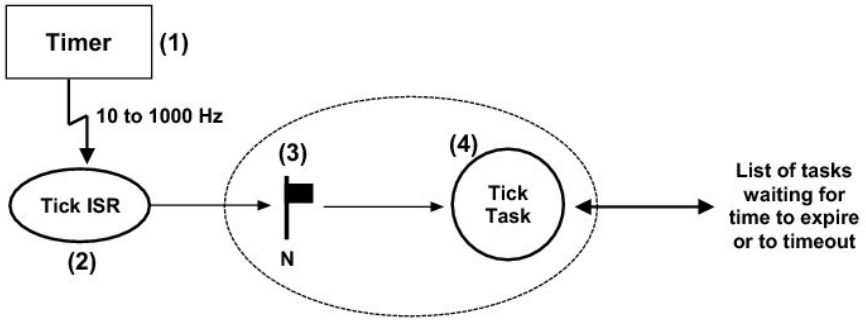


Figure - Tick ISR and Tick Task relationship

(1) A hardware timer is generally used and configured to generate an interrupt at a rate between 10 and 1000 Hz (see `OS_CFG_TICK_RATE_HZ` in `os_cfg_app.h`). This timer is generally called the *Tick Timer*. The actual rate to use depends on such factors as: processor speed, desired time resolution, and amount of allowable overhead to handle the tick timer, etc.

The tick interrupt does not have to be generated by a timer and, in fact, it can come from other regular time sources such as the power-line frequency (50 or 60 Hz), which are

known to be fairly accurate over long periods of time.

(2) Assuming CPU interrupts are enabled, the CPU accepts the tick interrupt, preempts the current task, and vectors to the tick ISR. The tick ISR must call `OSTimeTick()` (see `os_time.c`), which accomplishes most of the work needed by µC/OS-III. The tick ISR then clears the timer interrupt (and possibly reloads the timer for the next interrupt). However, some timers may need to be taken care of prior to calling `OSTimeTick()` instead of after as shown below.

```
void  TickISR (void)
         {
             OSTimeTick();
             /* Clear tick interrupt source             */
             /* Reload the timer for the next interrupt */
         }
```

Listing - Handle timer source after OSTimeTick()

or,

µC/OS-III may need to place literally hundreds of tasks (if an application has that many tasks) in the tick list. The tick list is implemented such that it takes as little CPU time as possible to determine if time has expired for those tasks placed in the tick list and, possibly makes those tasks ready-to-run. In fact, the tick management has been completely redesigned as of V3.04.00 and the tick list is actually split into two separate ‘delta-lists’ as shown in the figure below.

Copyright 2015 Micrium Inc.

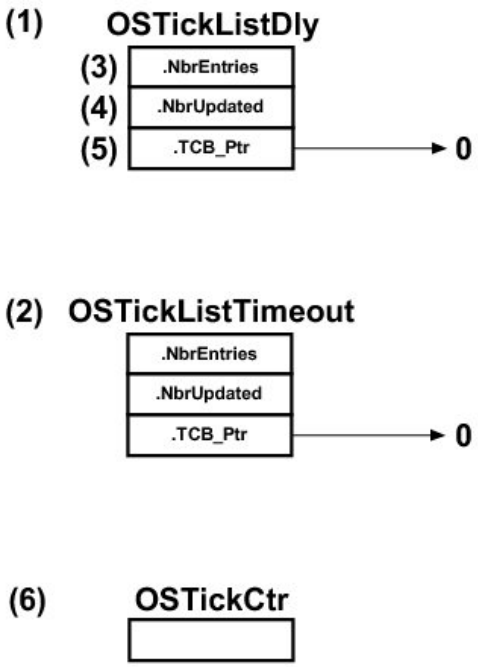


Figure - Empty Tick Lists

(1) This list contains all the tasks that are waiting for time to expire and thus contains all the `OS_TCB`s for the tasks that have called `OSTimeDly()` or `OSTimeDlyHMSM()`.

(2) This list contains all the tasks that are pending on an object with a timeout (i.e. they called `OS???Pend()` and specified a non-zero timeout value).

(3) The `.NbrEntries` field in each list contains the current number of entries in the list and is thus updated each time an `OS_TCB` is either inserted or removed from the list. This field is useful when debugging and serves no other purpose otherwise. This field is only present when `OS_CFG_DBG_EN` is set to 1 in `os_cfg.h`.

(4) The `.NbrUpdated` field in each list contains the number of `OS_TCB`s updated whenever a tick occurs. This field is useful when debugging and serves no other purpose otherwise. This field is only present when `OS_CFG_DBG_EN` is set to 1 in `os_cfg.h`.

(5) The `.TCB_List` contains a pointer to a doubly linked list of `OS_TCB`s that are placed in the list.

Copyright 2015 Micrium Inc.

(6) `OSTickCtr` is incremented by `OS_TickTask()` each time the task is signaled by the tick ISR. `OSTickCtr` simply keeps track of the number of ticks since startup and its value can be ready by calling `OSTimeGet()` or changed by calling `OSTimeSet()`.

Tasks are automatically inserted in the proper tick list when the application programmer calls a `OSTimeDly???()` function, or when an `OS???Pend()` call is made with a non-zero timeout value. If the users application does not need time delays or timeouts, the user can remove the Tick Task from µC/OS-III, see the µC-OS-III Configuration Manual for more information.

Example

Using an example to illustrate the process of inserting a task in the delayed tick list. Here, we assume that the `OSTickListDly` list is completely empty as shown in the figure below. A task is placed in the tick list when `OSTimeDly()` is called and let's assume `OSTimeDly()` is called as follows:

```
:
            OSTimeDly(10, OS_OPT_TIME_DLY, &err);
            :
```

Referring to the µC-OS-III API Reference Manual that this action indicates that µC/OS-III has to delay the current task for 10 ticks. Since this is the first task inserted in the tick list, the `.TickNextPtr` and `.TickPrevPtr` of the task's `OS_TCB` both point to `NULL`.

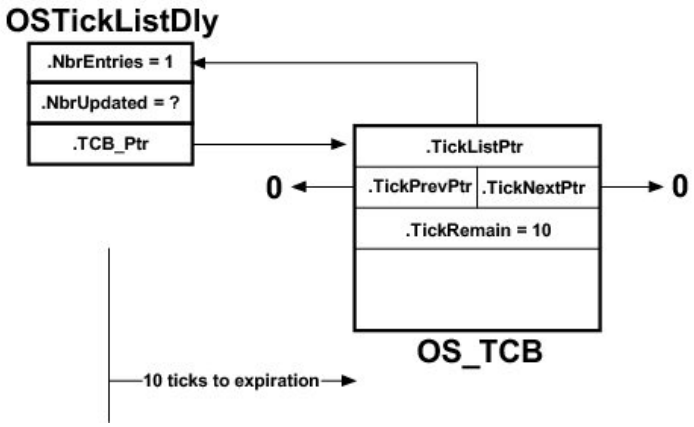


Figure - Inserting a task in the delayed tasks tick list

OSTimeDly() takes care of a few other details. Specifically, the task is removed from µC/OS-III's ready list (described in The Ready List) since the task is no longer eligible to run (because it is waiting for time to expire). Also, the scheduler is called because µC/OS-III will need to run the next most important ready-to-run task.

If the next task to run also happens to call OSTimeDly() "before" the next tick arrives and calls OSTimeDly() as follows:

```
:
            OSTimeDly(7, OS_OPT_TIME_DLY, &err);
            :
```

µC/OS-III will place this new task at the head of the list and replace the "remaining" counts for the first task to 3. In other words, the timeout for the first task is the sum of the two values (7+3). When the tick task runs, it only needs to decrement the .TickRemain of the OS_TCB which is at the head of the list, and worry about the next OS_TCB in the list three ticks later.

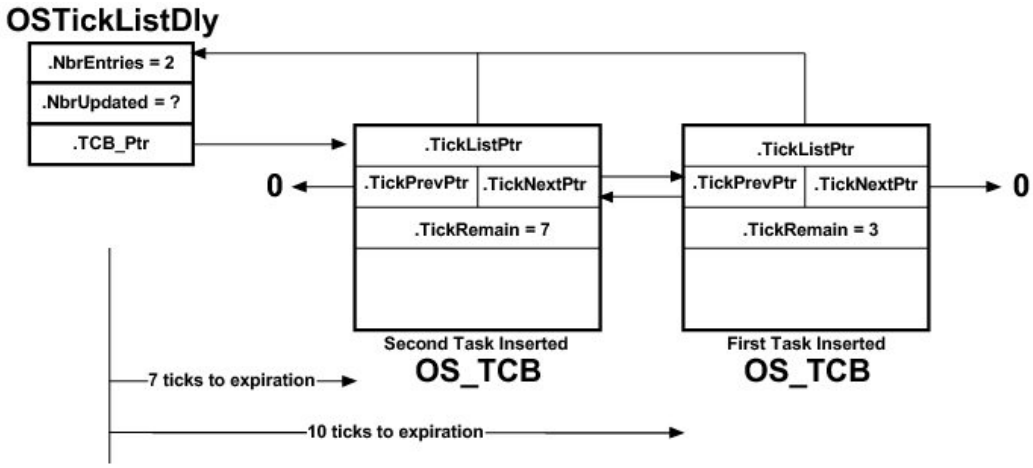


Figure - Inserting a second task in the delayed tasks tick list

When the tick task executes (see OS_TickTask() in os_tick.c), it starts by incrementing OSTickCtr and, if the delayed tasks list is not empty, decrements **only** the .TickRemain of the OS_TCB at the head of the list. When .TickRemain reaches zero, the OS_TCB is removed from the list and placed in the ready-list. Then, the .TickRemain field of the next OS_TCB is examined and if the .TickRemain field for that task is also zero, that task is also placed in the ready list. µC/OS-III continues like this until the .TickRemain of an OS_TCB has a non-zero value or, it reaches the end of the list.

Copyright 2015 Micrium Inc.

The .NbrUpdated field of the tick list contains the number of OS_TCBs removed from the list whenever a tick occurs. Of course, if .TickRemain is non-zero, .NbrUpdated would also be zero because on that specific tick, no task would be made ready-to-run.

The list of tasks that are waiting for timeouts works exactly the same as the delayed task list. The reason there are two lists is to reduce the amount of time in a critical section.

Dynamic Tick

µC/OS-III offers a mode where the Tick task is not run periodically but behaves as it does. In other words, as far as your application is concerned, it will not know the difference. The benefit of this mode come apparent when you are concerned about power consumption such as with battery operated devices. Dynamic Tick mode has the advantage of not waking up the Tick task unless it has to.

Without Dynamic Tick, if a task needs to sleep for one second, as done by calling OSTimeDlyHMSM(), and OS_CFG_TICK_RATE_HZ is set to 1000, the Tick task would run 999 times for nothing before finally waking up the delayed task on it's 1000th run. This can impose a large overhead in terms of power usage. The standard method used for the Tick task is to wake up OS_CFG_TICK_RATE_HZ times per second by the Tick ISR. Then, the tick task increments OSTickCtr by one and updates the timeout and delay tick lists, reading all tasks that have an expired delay or timeout.

It would be far more efficient to actually run the Tick task once the 1 second delay has elapsed. In order to support this new mode, we had to make some changes to the way the Tick task and ISR interact with each other. In the Dynamic ticking mode, the Tick task is awakened when the smallest needed delay of any tasks of either tick lists has elapsed. This minimum delay, called the Tick Step, is calculated every time a task needs to be delayed, every time a task pends on a kernel object with a timeout and every time the Tick task is run. This Tick Step is applied to the Dynamic tick timer by calling the BSP_OS_TickNextSet() function.
This guarantees that the Tick ISR will be called when at least Tick Step ticks have elapsed. The Tick ISR will then notify the Tick task, using OSTimeDynTick(), that Tick Step ticks have indeed elapsed. Now, instead of increment OSTickCtr by one and updating the tick lists, the Tick task will add the Tick Step to OSTickCtr and update the tick lists accordingly. After updating these lists, the Tick task will determine the new Tick Step based on the updated tick lists and apply this step to the Dynamic tick timer. This ensures that by setting a variable delay in the Dynamic tick timer, the Tick task will be awakened by the Tick ISR only when necessary. Note that to use the Dynamic Tick feature, the µC/OS-III port developer has

Copyright 2015 Micrium Inc.

to implement the Dynamic Tick API described in µC/OS-III Port and `OS_CFG_DYN_TICK_EN` needs to be set to `DEF_ENABLED` in `os_cfg.h`.

You should note that there are practical limitations in using the Dynamic Tick mode. For one thing, if you use a 16-bit timer that increments at a rate of 1 MHz then the maximum reload time for the timer would be 65,535 milliseconds and thus, you could not rely on such a timer to provide a 1 second delay. In this case, you probably would want the maximum Tick Step to be in increments of 65 milliseconds. However, I'd probably set the maximum increment to 50 making this a clean multiple of 1 second.

Copyright 2015 Micrium Inc.

The Statistic Task

µC/OS-III contains an internal task that provides such run-time statistics as overall CPU utilization (0.00 to 100.00%), per-task CPU utilization (0.00 to 100.00%), and per-task stack usage. As of V3.03.00, CPU utilization is represented as a integer from 0 to 10,000 (0.00% to 100.00%). Prior to V3.03.00, CPU utilization was represented an integer ranging from 0 to 100.

The statistic task is optional in a µC/OS-III application and its presence is controlled by a compile-time configuration constant `OS_CFG_STAT_TASK_EN` defined in `os_cfg.h`. Specifically, the code is included in the build when `OS_CFG_STAT_TASK_EN` is set to `DEF_ENABLED`.

Also, the priority of this task and the location and size of the statistic task's stack is configurable via `OS_CFG_STAT_TASK_PRIO` declared in `os_cfg_app.h` ().

If the application uses the statistic task, it should call `OSStatTaskCPUUsageInit()` from the first, and only application task created in the `main()` function as shown in the listing below. The startup code should create only one task before calling `OSStart()`. The single task created is, of course, allowed to create other tasks, but only after calling `OSStatTaskCPUUsageInit()`.

Copyright 2015 Micrium Inc.

```
void main (void)                    (1)
{
    OS_ERR  err;
    :
    OSInit(&err);                   (2)
    if (err != OS_ERR_NONE) {
        /* Something wasn't configured properly, µC/OS-III not properly initialized  */
        /* ... 'err' will tell you the cause (see os.h)                              */
    }
    /* (3) Create ONE task (we'll call it AppTaskStart() for sake of discussion)     */
    :
    OSStart(&err);                  (4)
}


void AppTaskStart (void *p_arg)
{
    OS_ERR  err;
    :
    /* (5) Initialize the tick interrupt                                             */
#if OS_CFG_STAT_TASK_EN == DEF_ENABLED
    OSStatTaskCPUUsageInit(&err);   (6)
    OSStatReset(&err)               (7)
#endif
    :
    /* (8) Create other tasks                                                        */
    while (DEF_ON) {
        /* AppTaskStart() body                                                       */
    }
}
```

Listing - Proper startup for computing CPU utilization

(1) The C compiler should start up the CPU and bring it to `main()` as is typical in most C applications.

(2) `main()` calls `OSInit()` to initialize µC/OS-III. It is assumed that the statistics task was enabled by setting `OS_CFG_STAT_TASK_EN` to `DEF_ENABLED` in `os_cfg.h`. You should always examine µC/OS-III's returned error code to make sure the call was done properly. Refer to `os.h` for a list of possible errors, `OS_ERR_???`.

(3) As the comment indicates, you should create a single task called `AppTaskStart()` in the example (its name is left to the creator's discretion). When creating this task, give it a fairly high priority (do not use priority 0 since it's reserved for µC/OS-III).

Normally, µC/OS-III allows the user to create as many tasks as are necessary prior to calling `OSStart()`. However, when the statistic task is used to compute overall CPU utilization, it is necessary to create only one task.

Copyright 2015 Micrium Inc.

(4) You need to call `OSStart()` to let µC/OS-III start the highest-priority task which, in our case is `AppTaskStart()`. At this point, there should be either four (4) to six (6) tasks created depending on configuration option: `OS_IdleTask()`, `OS_TickTask()`, `OS_StatTask()`, `OS_TmrTask()` (optional), `OS_IntQTask()` (optional) and now `AppTaskStart()`.

(5) The start task should then configure and enable tick interrupts. This most likely requires that the user initialize the hardware timer used for the clock tick and have it interrupt at the rate specified by `OS_CFG_TICK_RATE_HZ` (see `os_cfg_app.h`). Additionally, Micriμm provides sample projects that include a basic board-support package (BSP). The BSP initializes many aspects of the CPU as well as the periodic time source required by µC/OS-III. If available, the user may utilize BSP services by calling `BSP_Init()` from the startup task. After this point, no further time source initialization is required by the user.

(6) `OSStatTaskCPUUsageInit()` is called to determine the maximum value that `OSStatTaskCtr` (see `OS_IdleTask()`) can count up to for `1/OS_CFG_STAT_TASK_RATE_HZ` second when there are no other tasks running in the system (apart for the other µC/OS-III tasks). For example, if the system does not contain an application task and `OSStatTaskCtr` counts from 0 to 10,000,000 for `1/OS_CFG_STAT_TASK_RATE_HZ` second, when adding tasks, and the test is redone every `1/OS_CFG_STAT_TASK_RATE_HZ` second, the `OSStatTaskCtr` will not reach 10,000,000 and actual CPU utilization is determined as follows:

$$CPU\_Utilization_{\%} = \left(100 - \frac{100 \times OSStatTaskCtr}{OSStatTaskCtrMax}\right)$$

For example, if when redoing the test, `OSStatTaskCtr` reaches 7,500,000 the CPU is busy 25% of its time running application tasks:

$$25\% = \left(100 - \frac{100x7{,}500{,}000}{10{,}000{,}000}\right)$$

(7) You should call this function to reset the statistics counter so that you are not getting incorrect values from the startup code.

(8) `AppTaskStart()` can then create other application tasks as needed.

As previously described, µC/OS-III stores run-time statistics for each task in each task's

OS_TCB.

OS_StatTask() also computes stack usage of all created tasks by calling OSTaskStkChk() (see os_task.c) and stores the return values of this function (free and used stack space) in the .StkFree and .StkUsed field of the task's OS_TCB, respectively.

The Timer Task

µC/OS-III provides timer services to the application programmer and this code is found in `os_tmr.c`.

The timer task is optional in a µC/OS-III application and its presence is controlled by the compile-time configuration constant `OS_CFG_TMR_EN` defined in `os_cfg.h`. Specifically, the code is included in the build when `OS_CFG_TMR_EN` is set to `DEF_ENABLED`.

Timers are countdown counters that perform an action when the counter reaches zero. The action is provided by the user through a callback function. A callback function is a function that the user declares and that will be called when the timer expires. The callback can thus be used to turn on or off a light, a motor, or perform whatever action needed. It is important to note that the callback function is called from the context of the timer task. The application programmer may create an unlimited number of timers (limited only by the amount of available RAM). Timer management is fully described in Timer Management and the timer services available to the application programmer are described in the µC-OS-III API Reference .

`OS_TmrTask()` is a task created by µC/OS-III (this assumes setting `OS_CFG_TMR_EN` to `DEF_ENABLED` in `os_cfg.h`) and its priority is configurable by the user through µC/OS-III's configuration constant `OS_CFG_TMR_TASK_PRIO` found in `os_cfg_app.h`. `OS_TmrTask()` is typically set to a low to medium priority.

`OS_TmrTask()` is a periodic task using the same interrupt source that was used to generate clock ticks. However, timers are generally updated at a slower rate (i.e., typically 10 Hz). This is accomplished by dividing down the timer tick rate in software. In other words, if the tick rate is 1000 Hz and the desired timer rate is 10 Hz, the timer task will be signaled every 100th tick interrupt as shown in the figure below.

Copyright 2015 Micrium Inc.

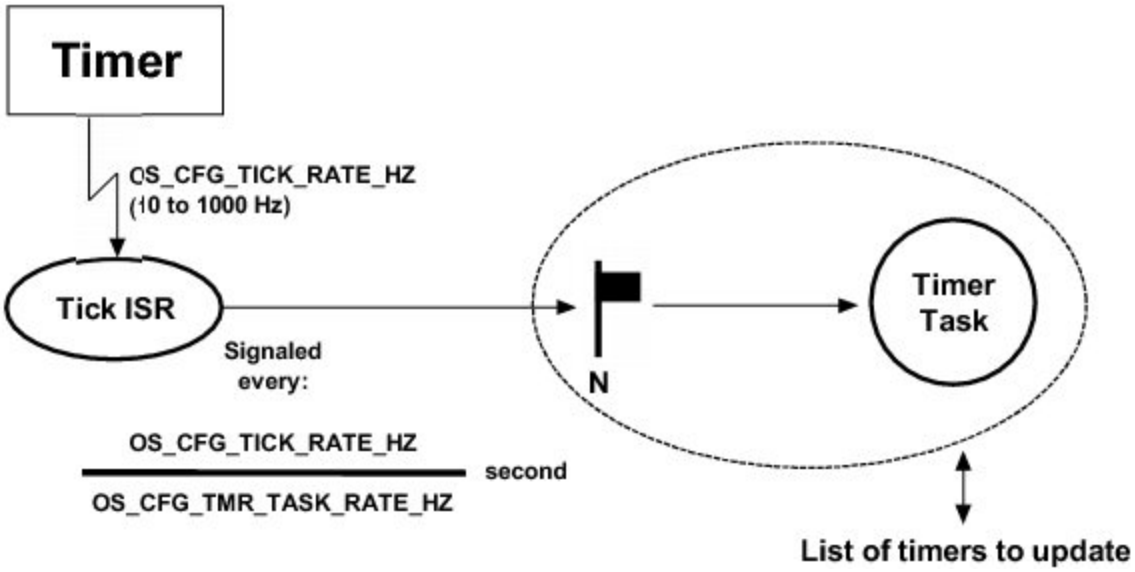


Figure - Tick ISR and Timer Task relationship

The ISR Handler Task

When setting the compile-time configuration constant `OS_CFG_ISR_POST_DEFERRED_EN` in `os_cfg.h` to `DEF_ENABLED`, µC/OS-III creates a task (called `OS_IntQTask()`) responsible for "deferring" the action of OS post service calls from ISRs.

As described in Critical Sections, µC/OS-III manages critical sections either by disabling/enabling interrupts, or by locking/unlocking the scheduler. If selecting the latter method (i.e., setting `OS_CFG_ISR_POST_DEFERRED_EN` to `DEF_ENABLED`), µC/OS-III "post" functions called from interrupts are not allowed to manipulate such internal data structures as the ready list, pend lists, and others.

When an ISR calls one of the "post" functions provided by µC/OS-III, a copy of the data posted and the desired destination is placed in a special "holding" queue. When all nested ISRs complete, µC/OS-III context switches to the ISR handler task (`OS_IntQTask()`), which "re-posts" the information placed in the holding queue to the appropriate task(s). This extra step is performed to reduce the amount of interrupt disable time that would otherwise be necessary to remove tasks from wait lists, insert them in the ready list, and perform other time-consuming operations.

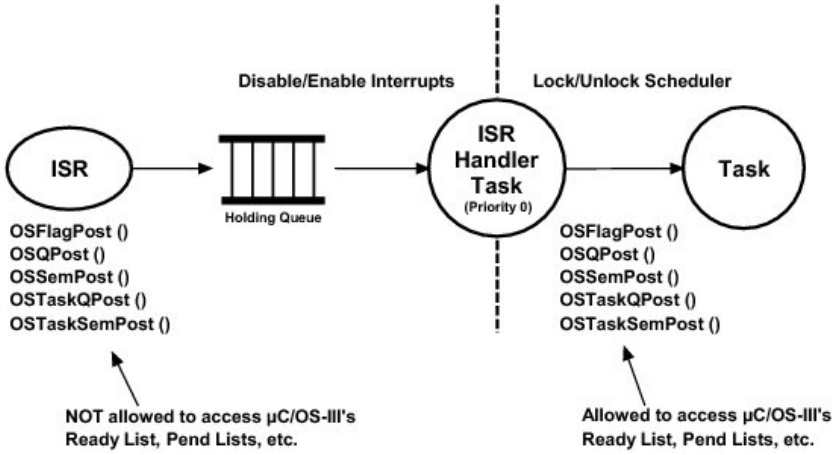


Figure - ISR Handler Task

`OS_IntQTask()` is created by µC/OS-III and always runs at priority 0 (i.e., the highest priority). If `OS_CFG_ISR_POST_DEFERRED_EN` is set to `DEF_ENABLED`, no other task will be allowed to use priority 0.

Copyright 2015 Micrium Inc.

The Ready List

Tasks that are ready to execute are placed in the Ready List. The ready list consists of two parts: a bitmap containing the priority levels that are ready and a table containing pointers to all the tasks ready.

In this section:

- Priority Levels
- About the Ready List
- Adding Tasks to the Ready List

Copyright 2015 Micrium Inc.

Priority Levels

The figures below show the bitmap of priorities that are ready. The “width” of the table depends on the data type `CPU_DATA` (see `cpu.h`), which can either be 8-, 16- or 32-bits. The width depends on the processor used.

µC/OS-III allows up to `OS_CFG_PRIO_MAX` different priority levels (see `os_cfg.h`). In µC/OS-III, a low-priority number corresponds to a high-priority level. Priority level zero (0) is thus the highest priority level. Priority `OS_CFG_PRIO_MAX-1` is the lowest priority level. µC/OS-III uniquely assigns the lowest priority to the idle task and thus, no other tasks are allowed at this priority level. If there are tasks that are ready-to-run at a given a priority level, then its corresponding bit is set (i.e., 1) in the bitmap table. Notice in the figures below that “priority levels” are numbered from left to right and, the priority level increases (moves toward lower priority) with an increase in table index. The order was chosen to be able to use a special instruction called Count Leading Zeros (CLZ), which is found on many modern processors. This instruction greatly accelerates the process of determining the highest priority level.

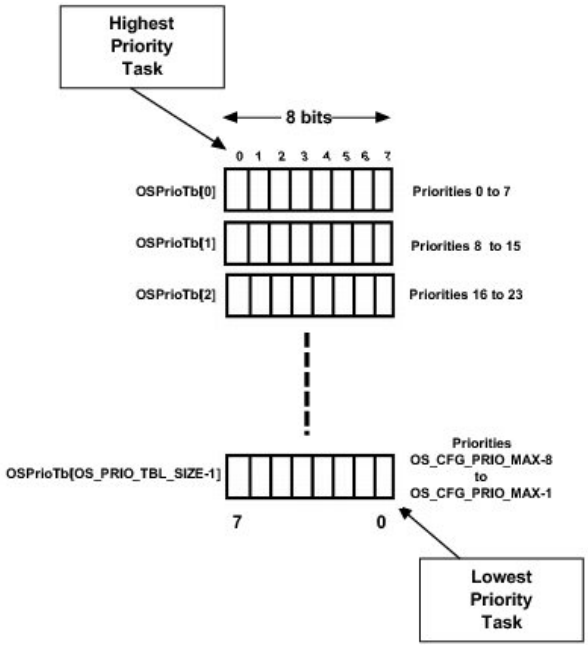


Figure - CPU_DATA declared as a CPU_INT08U

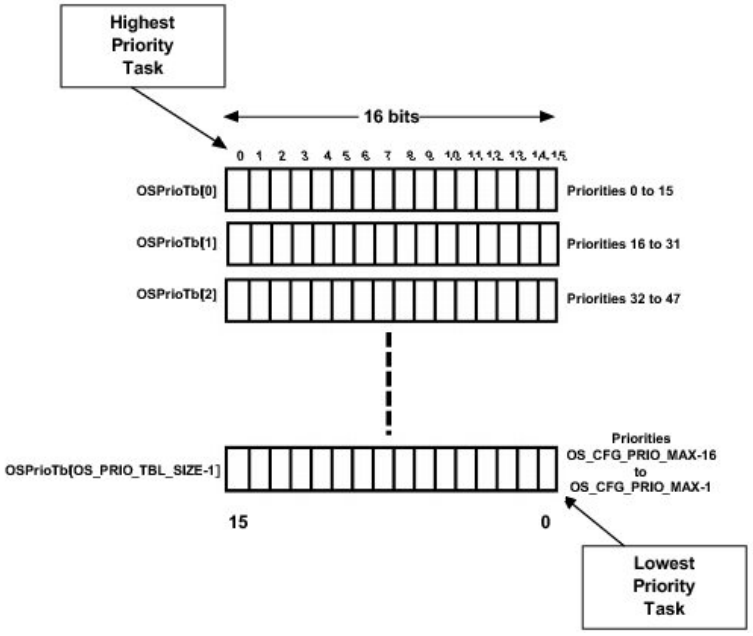


Figure - CPU_DATA declared as a CPU_INT16U

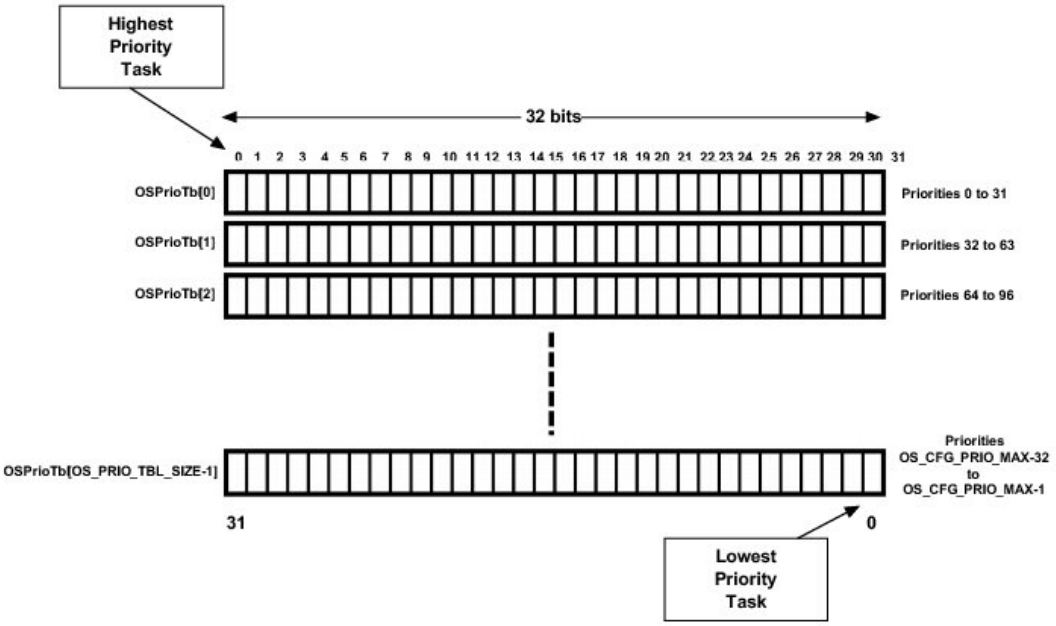


Figure - CPU_DATA declared as a CPU_INT32U

`os_prio.c` contains the code to set, clear, and search the bitmap table. These functions are internal to µC/OS-III and are placed in `os_prio.c` to allow them to be optimized in assembly language by replacing `os_prio.c` with an assembly language equivalent `os_prio.asm`, if necessary.

Copyright 2015 Micrium Inc.

Function	Description
OS_PrioGetHighest()	Find the highest priority level
OS_PrioInsert()	Set bit corresponding to priority level in the bitmap table
OS_PrioRemove()	Clear bit corresponding to priority level in the bitmap table

Table - Priority Level access functions

To determine the highest priority level that contains ready-to-run tasks, the bitmap table is scanned until the first bit set in the lowest bit position is found using OS_PrioGetHighest(). The code for this function is shown in the listing below.

```
OS_PRIO  OS_PrioGetHighest (void)
         {
             CPU_DATA  *p_tbl;
             OS_PRIO    prio;


             prio  = (OS_PRIO)0;
             p_tbl = &OSPrioTbl[0];
             while (*p_tbl == (CPU_DATA)0) {               (1)
                 prio += DEF_INT_CPU_NBR_BITS;             (2)
                 p_tbl++;
             }
             prio += (OS_PRIO)CPU_CntLeadZeros(*p_tbl);    (3)
             return (prio);
         }
```

Listing - Finding the highest priority level

(1) OS_PrioGetHighest() scans the table from OSPrioTbl[0] until a non-zero entry is found. The loop will always terminate because there will always be a non-zero entry in the table because of the idle task.

(2) Each time a zero entry is found, we move to the next table entry and increment "prio" by the width (in number of bits) of each entry. If each entry is 32-bits wide, "prio" is incremented by 32.

(3) Once the first non-zero entry is found, the number of "leading zeros" of that entry is simply added and we return the priority level back to the caller. Counting the number of zeros is a CPU-specific function so that if a particular CPU has a built-in CLZ

Copyright 2015 Micrium Inc.

instruction, it is up to the implementer of the CPU port to take advantage of this feature. If the CPU used does not provide that instruction, the functionality must be implemented in C.

The function `CPU_CntLeadZeros()` simply counts how many zeros there are in a `CPU_DATA` entry starting from the left (i.e., most significant bit). For example, assuming 32 bits, `0xF0001234` results in 0 leading zeros and `0x00F01234` results in 8 leading zeros.

At first view, the linear path through the table might seem inefficient. However, if the number of priority levels is kept low, the search is quite fast. In fact, there are several optimizations to streamline the search. For example, if using a 32-bit processor and you are satisfied with limiting the number of different priority levels to 64, the above code can be optimized as shown in the listing below. In fact, some processors have built-in “Count Leading Zeros” instructions and thus, the code can even be written with just a few lines of assembly language instead of C. Remember that with µC/OS-III, 64 priority levels does not mean that the user is limited to 64 tasks since with µC/OS-III, any number of tasks are possible at a given priority level (except `0` and `OS_CFG_PRIO_MAX-1`).

```
OS_PRIO  OS_PrioGetHighest (void)
         {
             OS_PRIO  prio;


             if (OSPrioTbl[0] != (OS_PRIO_BITMAP)0) {
                 prio = OS_CntLeadZeros(OSPrioTbl[0]);
             } else {
                 prio = OS_CntLeadZeros(OSPrioTbl[1]) + 32;
             }
             return (prio);
         }
```

Listing - Finding the highest priority level within 64 levels

About the Ready List

Tasks that are ready-to-run are placed in the Ready List. As shown in the figure below, the ready list is an array (`OSRdyList[]`) containing `OS_CFG_PRIO_MAX` entries, with each entry defined by the data type `OS_RDY_LIST` (see `os.h`). An `OS_RDY_LIST` entry consists of three fields: `.Entries`, `.TailPtr` and `.HeadPtr`.

`.Entries` contains the number of ready-to-run tasks at the priority level corresponding to the entry in the ready list. `.Entries` is set to zero (0) if there are no tasks ready-to-run at a given priority level.

`.TailPtr` and `.HeadPtr` are used to create a doubly linked list of all the tasks that are ready at a specific priority. `.HeadPtr` points to the head of the list and `.TailPtr` points to its tail.

The “index” into the array is the priority level associated with a task. For example, if a task is created at priority level 5 then it will be inserted in the table at `OSRdyList[5]` if that task is ready-to-run.

The table below shows the functions that µC/OS-III uses to manipulate entries in the ready list. These functions are found in `os_core.c` and are internal to µC/OS-III so, the application code must never call them.

Function	Description
`OS_RdyListInit()`	Initialize the ready list to “empty” (see the figure below)
`OS_RdyListInsert()`	Insert a TCB into the ready list
`OS_RdyListInsertHead()`	Insert a TCB at the head of the list
`OS_RdyListInsertTail()`	Insert a TCB at the tail of the list
`OS_RdyListMoveHeadToTail()`	Move a TCB from the head to the tail of the list
`OS_RdyListRemove()`	Remove a TCB from the ready list

Table - Ready List access functions

Copyright 2015 Micrium Inc.

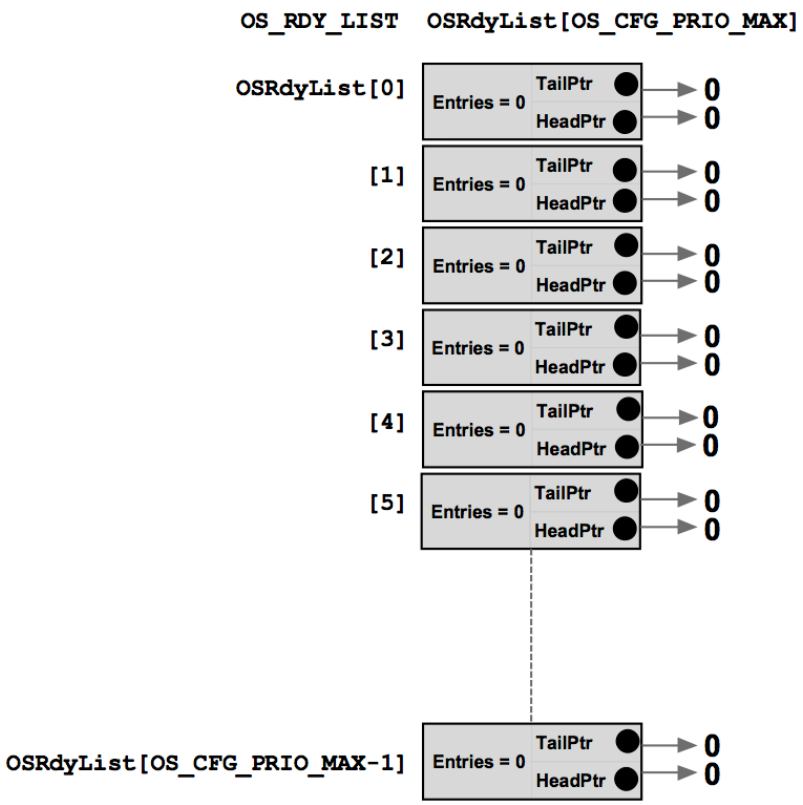


Figure - Empty Ready List

Assuming all internal µC/OS-III's tasks are enabled, the figure below shows the state of the ready list after calling `OSInit()` (i.e., µC/OS-III's initialization). It is assumed that each µC/OS-III task had a unique priority. With µC/OS-III, this does not have to be the case.

(1) There is only one entry in `OSRdyList[OS_CFG_PRIO_MAX-1]`, the idle task.

(2) The list points to OS_TCBs. Only relevant fields of the TCB are shown. The `.PrevPtr` and `.NextPtr` are used to form a doubly linked list of `OS_TCBs` associated to tasks at the same priority. For the idle task, these fields always point to `NULL`.

(3) `Priority 0` is reserved to the ISR handler task when `OS_CFG_ISR_DEFERRED_EN` is set to `DEF_ENABLED` in `os_cfg.h`. In this case, this is the only task that can run at priority 0.

Copyright 2015 Micrium Inc.

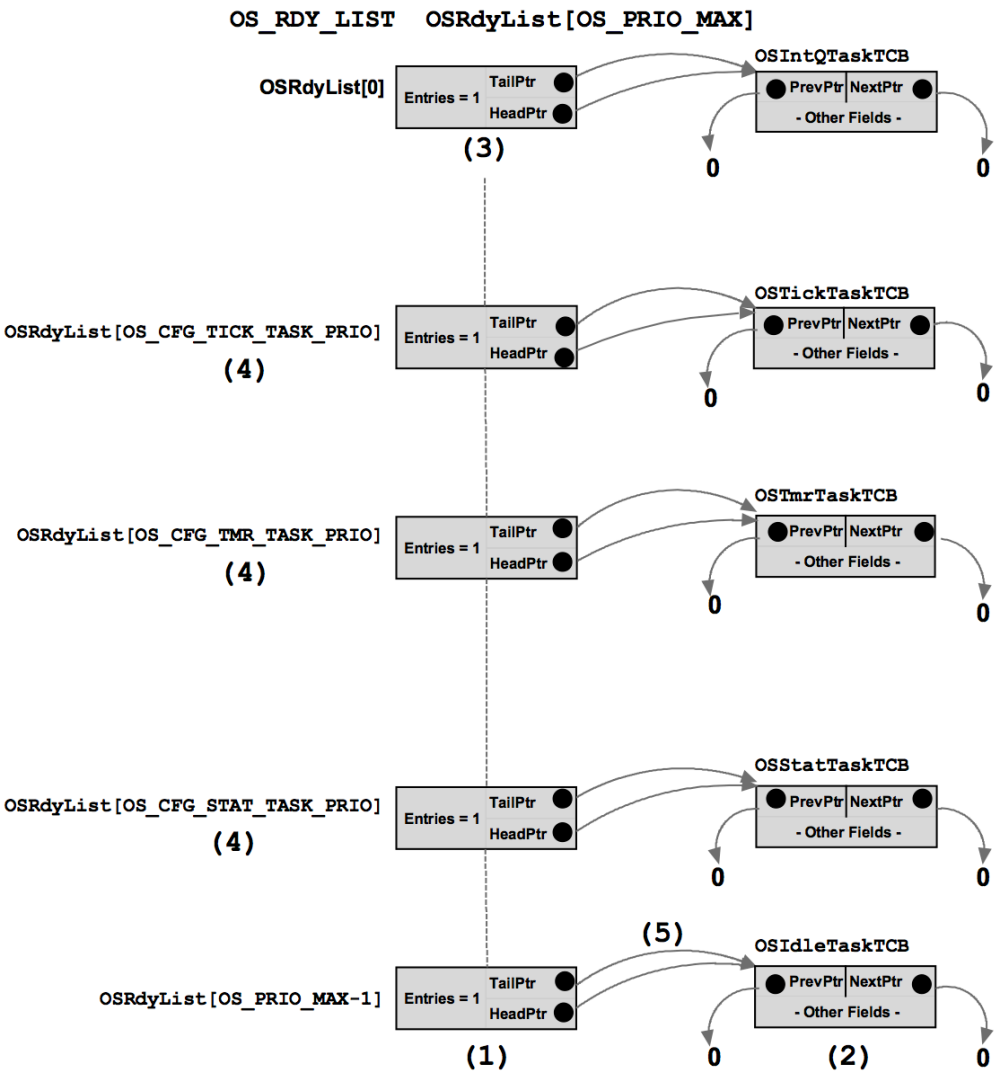


Figure - Ready List after calling OSInit()

(1) The tick task and the other three optional tasks have their own priority level, as shown. Typically, you would set the priority of the tick task higher than the timer task and, the timer task higher in priority than the statistic task.

(2) Both the tail and head pointers point to the same TCB when there is only one TCB at a given priority level.

Copyright 2015 Micrium Inc.

Adding Tasks to the Ready List

Tasks are added to the ready list by a number of µC/OS-III services. The most obvious service is `OSTaskCreate()`, which always creates a task in the ready-to-run state and adds the task to the ready list. As shown in Figure 6-6, we created a task, and specified a priority level where tasks already existed (two in this example) in the ready list at the desired priority level. `OSTaskCreate()` will then insert the new task at the end of the list of tasks at that priority level.

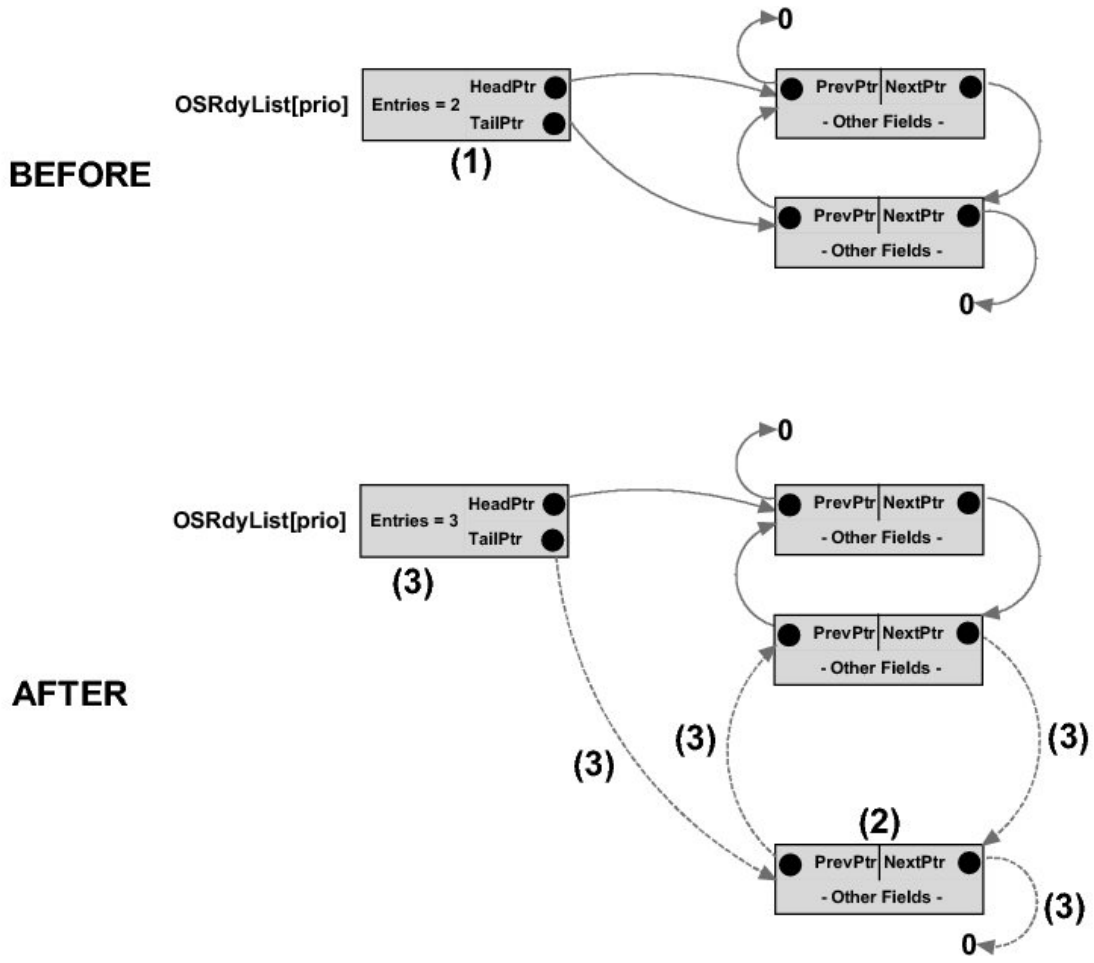


Figure - Inserting a newly created task in the ready list

(1) Before calling `OSTaskCreate()` (in this example), two tasks were in the ready list at priority "`prio`".

Copyright 2015 Micrium Inc.

(2) A new TCB is passed to `OSTaskCreate()` and, µC/OS-III initialized the contents of that TCB.

(3) `OSTaskCreate()` calls `OS_RdyListInsertTail()`, which links the new TCB to the ready list by setting up four pointers and also incrementing the `.Entries` field of `OSRdyList[prio]`. Not shown in Figure 6-6 is that `OSTaskCreate()` also calls `OS_PrioInsert()` to set the bit in the bitmap table. Of course, this operation is not necessary as there are already entries in the list at this priority. However, `OS_PrioInsert()` is a very fast call and thus it should not affect performance.

The reason the new TCB is added to the end of the list is that the current head of the list could be the task creator and it could be at the same priority. So, there is no reason to make the new task the next task to run. In fact, a task being made ready will be inserted at the tail of the list if the current task is at the same priority. However, if a task is being made ready at a different priority than the current task, it will be inserted at the head of the list.

Copyright 2015 Micrium Inc.

Scheduling

The scheduler, also called the dispatcher, is a part of µC/OS-III responsible for determining which task runs next. µC/OS-III is a *preemptive, priority-based kernel*. As we have seen, each task is assigned a priority based on its importance. The priority for each task depends on the application, and µC/OS-III supports multiple tasks at the same priority level.

The word preemptive means that when an event occurs, and that event makes a more important task ready-to-run, then µC/OS-III will immediately give control of the CPU to that task. Thus, when a task signals or sends a message to a higher-priority task, the current task is suspended and the higher-priority task is given control of the CPU. Similarly, if an Interrupt Service Routine (ISR) signals or sends a message to a higher priority task, when the message has been sent, the interrupted task remains suspended, and the new higher priority task resumes.

Copyright 2015 Micrium Inc.

Preemptive Scheduling

µC/OS-III handles event posting from interrupts using two different methods: Direct and Deferred Post. These will be discussed in greater detail in Interrupt Management. From a scheduling point of view, the end result of the two methods is the same; the highest priority ready task will receive the CPU as shown in the figures below.

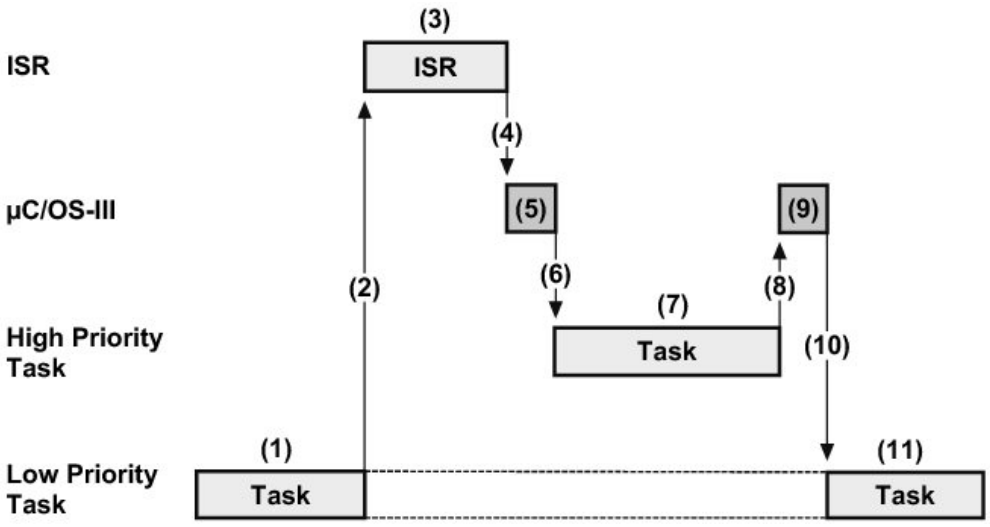


Figure - Preemptive scheduling – Direct Method

(1) A low priority task is executing, and an interrupt occurs.

(2) If interrupts are enabled, the CPU vectors (i.e., jumps) to the ISR that is responsible for servicing the interrupting device.

(3) The ISR services the device and signals or sends a message to a higher-priority task waiting to service this device. This task is thus ready-to-run.

(4) When the ISR completes its work it makes a service call to µC/OS-III.

(5)

(6) Since there is a more important ready-to-run task, µC/OS-III decides to not return to the interrupted task but switches to the more important task. See Context Switching for

details on how this works.

(7)

(8) The higher priority task services the interrupting device and, when finished, calls µC/OS-III asking it to wait for another interrupt from the device.

(9)

(10) µC/OS-III blocks the high-priority task until the next time the device needs servicing. Since the device has not interrupted a second time, µC/OS-III switches back to the original task (the one that was interrupted).

(11) The interrupted task resumes execution, exactly at the point where it was interrupted.

The figure below shows that µC/OS-III performs a few extra steps when it is configured for the Deferred Post method. Notice that the end results is the same; the high-priority task preempts the low-priority one.

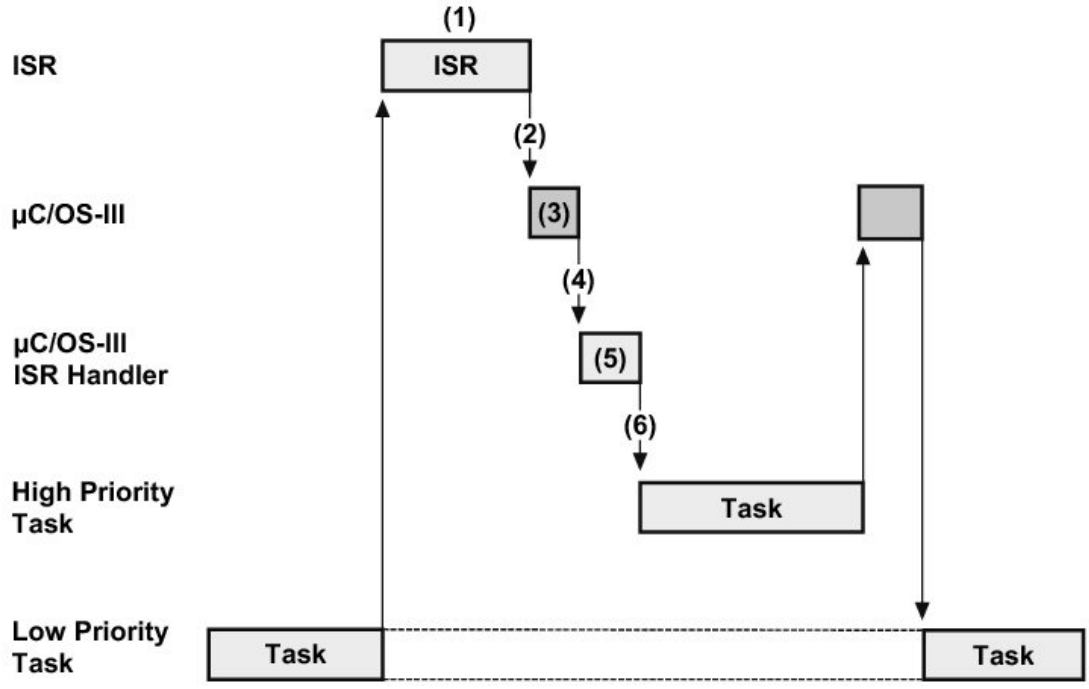


Figure - Preemptive scheduling – Deferred Post Method

(1) The ISR services the device and, instead of signaling or sending the message to the task, µC/OS-III (through the POST call) places the post call into a special queue and makes a very high-priority task (actually the highest-possible priority) ready-to-run. This task is called the *ISR Handler Task*.

(2) When the ISR completes its work, it makes a service call to µC/OS-III.

(3)

(4) Since the ISR made the ISR Handler Task ready-to-run, µC/OS-III switches to that task.

(5)

(6) The ISR Handler Task then removes the post call from the message queue and reissues the post. This time, however, it does it at the task level instead of the ISR level. The reason this extra step is performed is to keep interrupt disable time as small as possible. See Interrupt Management to find out more on the subject. When the queue is emptied, µC/OS-III removes the ISR Handler Task from the ready list and switches to the task that was signaled or sent a message.

Copyright 2015 Micrium Inc.

Scheduling Points

Scheduling occurs at scheduling points and nothing special must be done in the application code since scheduling occurs automatically based on the conditions described below.

A task signals or sends a message to another task:
This occurs when the task signaling or sending the message calls one of the post services, `OS???Post()`. Scheduling occurs towards the end of the `OS???Post()` call. Note that scheduling does not occur if one specifies (as part of the post call) to not invoke the scheduler (i.e., by setting the option argument to `OS_OPT_POST_NO_SCHED`).

A task calls `OSTimeDly()` or `OSTimeDlyHMSM()`:
If the delay is non-zero, scheduling always occurs since the calling task is placed in a list waiting for time to expire. Scheduling occurs as soon as the task is inserted in the wait list and this call will always result in a context switch to the next task that is ready-to-run at the same or lower priority than the task that called `OSTimeDly()` or `OSTimeDlyHMSM()`.

A task waits for an event to occur and the event has not yet occurred:
This occurs when one of the `OS???Pend()` functions are called. The task is placed in the wait list for the event and, if a non-zero timeout is specified, the task is also inserted in the list of tasks waiting to timeout. The scheduler is then called to select the next most important task to run.

If a task aborts a pend:
A task is able to abort the wait (i.e., pend) of another task by calling `OS???PendAbort()`. Scheduling occurs when the task is removed from the wait list for the specified kernel object.

If a task is created:
The newly created task may have a higher priority than the task's creator. In this case, the scheduler is called.

If a task is deleted:
When terminating a task, the scheduler is called if the current task is deleted.

If a kernel object is deleted:
If you delete an event flag group, a semaphore, a message queue, or a mutual exclusion

Copyright 2015 Micrium Inc.

semaphore, if tasks are waiting on the kernel object, those tasks will be made ready-to-run and the scheduler will be called to determine if any of the tasks have a higher priority than the task that deleted the kernel object. Those tasks will be notified that the kernel object was deleted.

A task changes the priority of itself or another task:
The scheduler is called when a task changes the priority of another task (or itself) and the new priority of that task is higher than the task that changed the priority.

A task suspends itself by calling `OSTaskSuspend()`:
The scheduler is called since the task that called `OSTaskSuspend()` is no longer able to execute. The suspended task must be resumed by another task.

A task resumes another task that was suspended by `OSTaskSuspend()`:
The scheduler is called if the resumed task has a higher priority than the task that calls `OSTaskResume()`.

At the end of all nested ISRs:
The scheduler is called at the end of all nested ISRs to determine whether a more important task is made ready-to-run by one of the ISRs. The scheduling is actually performed by `OSIntExit()` instead of `OSSched()`.

The scheduler is unlocked by calling OSSchedUnlock():
The scheduler is unlocked after being locked. You can lock the scheduler by calling `OSSchedLock()`. Note that locking the scheduler can be nested and the scheduler must be unlocked a number of times equal to the number of locks.

A task gives up its time quanta by calling `OSSchedRoundRobinYield()`:
This assumes that the task is running alongside with other tasks at the same priority and the currently running task decides that it can give up its time quanta and let another task run.

The user calls `OSSched()`:
The application code can call `OSSched()` to run the scheduler. This only makes sense if calling `OS???Post()` functions and specifying `OS_OPT_POST_NO_SCHED` so that multiple posts can be accomplished without running the scheduler on every post. However, in the above situation, the last post can be a post without the `OS_OPT_POST_NO_SCHED` option.

Copyright 2015 Micrium Inc.

Round-Robin Scheduling

When two or more tasks have the same priority, µC/OS-III allows one task to run for a predetermined amount of time (called a *Time Quanta*) before selecting another task. This process is called *Round-Robin Scheduling* or *Time Slicing*. If a task does not need to use its full time quanta it can voluntarily give up the CPU so that the next task can execute. This is called *Yielding*. µC/OS-III allows the user to enable or disable round robin scheduling at run time.

The figure below shows a timing diagram with tasks running at the same priority. There are three tasks that are ready-to-run at priority "X". For sake of illustration, the time quanta occurs every 4th clock tick. This is shown as a darker tick mark.

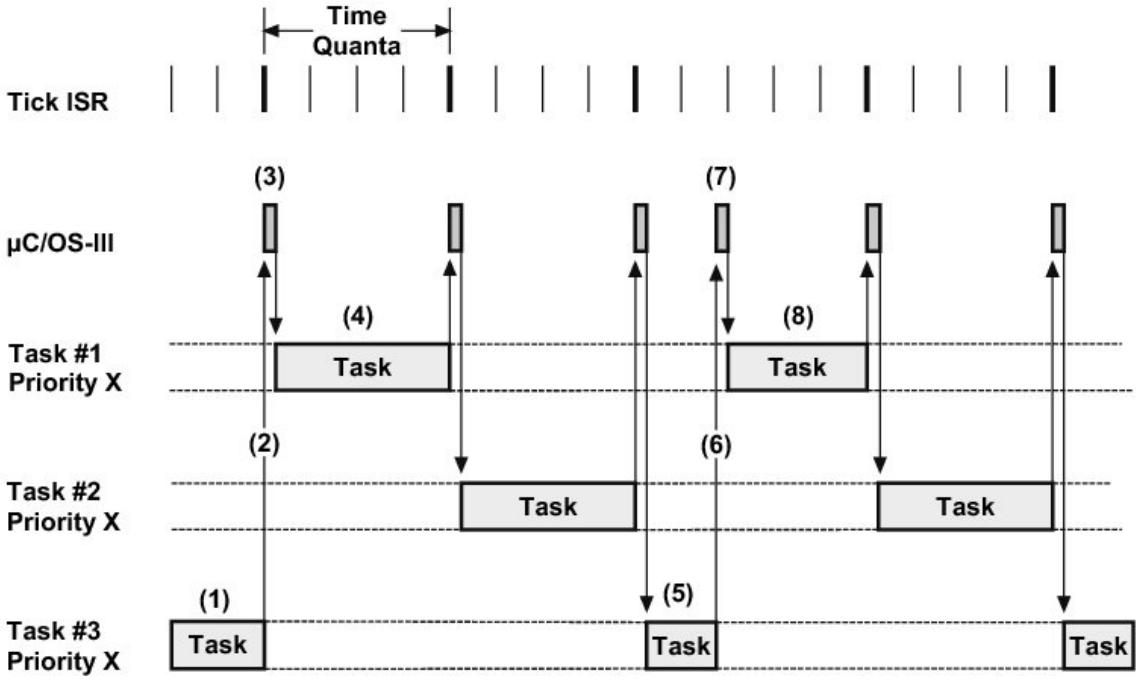


Figure - Round Robin Scheduling

(1) Task #3 is executing. During that time, a tick interrupt occurs but the time quanta have not expired yet for Task #3.

(2) On the 4th tick interrupt, the time quanta for Task #3 expire.

(3) µC/OS-III resumes Task #1 since it was the next task in the list of tasks at priority "X"

Copyright 2015 Micrium Inc.

that was ready-to-run.

(4) Task #1 executes until its time quanta expires (i.e., after four ticks).

(5)

(6)

(7) Here Task #3 executes but decides to give up its time quanta by calling the µC/OS-III function `OSSchedRoundRobinYield()`, which causes the next task in the list of tasks ready at priority "X" to execute. An interesting thing occurred when µC/OS-III scheduled Task #1. It reset the time quanta for that task to four ticks so that the next time quanta will expire four ticks from this point.

(8) Task #1 executes for its full time quanta.

µC/OS-III allows the user to change the default time quanta at run time through the `OSSchedRoundRobinCfg()` function (see Appendix A, "µC/OS-III API Reference"). This function also allows round robin scheduling to be enabled/disabled, and the ability to change the default time quanta.

µC/OS-III also enables the user to specify the time quanta on a per-task basis. One task could have a time quanta of 1 tick, another 12, another 3, and yet another 7, etc. The time quanta of a task is specified when the task is created. The time quanta of a task may also be changed at run time through the function `OSTaskTimeQuantaSet()`.

Copyright 2015 Micrium Inc.

Scheduling Internals

Scheduling is performed by two functions: `OSSched()` and `OSIntExit()`. `OSSched()` is called by task level code while `OSIntExit()` is called by ISRs. Both functions are found in `os_core.c`.

The figure below illustrates the two sets of data structures that the scheduler uses; the priority ready bitmap and the ready list as described in The Ready List.

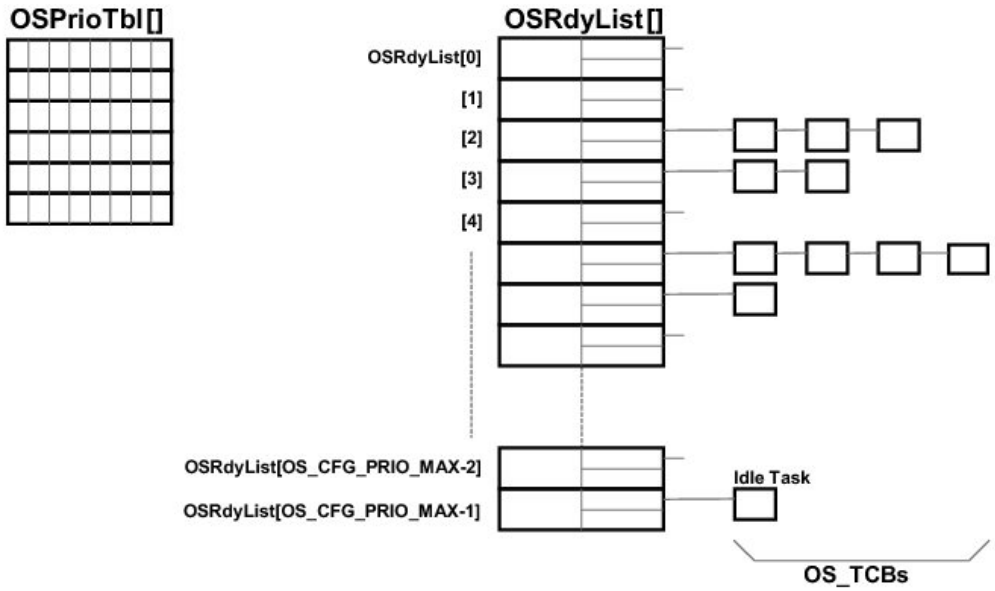


Figure - Priority ready bitmap and Ready list

OS_SchedRoundRobin()

When the time quanta for a task expires and there are multiple tasks at the same priority, µC/OS-III will select and run the next task that is ready-to-run at the current priority. `OS_SchedRoundRobin()` is the code used to perform this operation. `OS_SchedRoundRobin()` is either called by `OSTimeTick()` or `OS_IntQTask()`. `OS_SchedRoundRobin()` is found in `os_core.c`.

`OS_SchedRoundRobin()` is called by `OSTimeTick()` if you selected the Direct Method of posting (see Interrupt Management). `OS_SchedRoundRobin()` is called by `OS_IntQTask()` if you selected the Deferred Post Method of posting, described in Chapter 8.

The pseudo code for the round-robin scheduler is shown in Listing 7-3.

Copyright 2015 Micrium Inc.

```
void  OS_SchedRoundRobin (void)
{
    if (OSSchedRoundRobinEn != TRUE) {                          (1)
        return;
    }
    if (Time quanta counter > 0) {                              (2)
       Decrement time quanta counter;
    }
    if (Time quanta counter > 0) {
        return;
    }
    if (Number of OS_TCB at current priority level < 2) {       (3)
        return;
    }
    if (OSSchedLockNestingCtr > 0) {                            (4)
        return;
    }
    Move OS_TCB from head of list to tail of list;              (5)
    Reload time quanta for current task;                        (6)
}
```

Listing - OS_SchedRoundRobin() pseudocode

(1) `OS_SchedRoundRobin()` starts by making sure that round robin scheduling is enabled. Recall that to enable round robin scheduling, your code must call `OSSchedRoundRobinCfg()`.

(2) The time quanta counter, which resides inside the `OS_TCB` of the running task, is decremented. If the value is still non-zero then `OS_SchedRoundRobin()` returns.

(3) Once the time quanta counter reaches zero, `OS_SchedRoundRobin()` checks to see that there are other ready-to-run tasks at the current priority. If there are none, the function returns. Round robin scheduling only applies when there are multiple tasks at the same priority and the task doesn’t completes its work within its time quanta.

(4) `OS_SchedRoundRobin()` also returns if the scheduler is locked.

(5) Next, `OS_SchedRoundRobin()` moves the `OS_TCB` of the current task from the head of the ready list to the end.

(6) The time quanta for the task at the head of the list is loaded. Each task may specify its own time quanta when the task is created or through `OSTaskTimeQuantaSet()`. If you set the task time quanta to 0 then µC/OS-III assumes the default time quanta, which corresponds to the value in the variable `OSSchedRoundRobinDfltTimeQuanta`.

Copyright 2015 Micrium Inc.

Copyright 2015 Micrium Inc.

Context Switching

When µC/OS-III decides to run a different task (see Scheduling), it saves the current task's context, which typically consists of the CPU registers, onto the current task's stack and restores the context of the new task and resumes execution of that task. This process is called a *Context Switch*.

Context switching adds overhead. The more registers a CPU has, the higher the overhead. The time required to perform a context switch is generally determined by how many registers must be saved and restored by the CPU.

The context switch code is generally part of a processor's *port* of µC/OS-III. A port is the code needed to adapt µC/OS-III to the desired processor. This code is placed in special C and assembly language files: `os_cpu.h`, `os_cpu_c.c` and `os_cpu_a.asm`. Porting uC-OS-III provides more details on the steps needed to port µC/OS-III to different CPU architectures.

In this chapter, we will discuss the context switching process in generic terms using a fictitious CPU as shown in Figure 8-1. Our fictitious CPU contains 16 integer registers (R0 to R15), a separate ISR stack pointer, and a separate status register (SR). Every register is 32 bits wide and each of the 16 integer registers can hold either data or an address. The program counter (or instruction pointer) is R15 and there are two separate stack pointers labeled R14 and R14'. R14 represents a task stack pointer (TSP), and R14' represents an ISR stack pointer (ISP). The CPU automatically switches to the ISR stack when servicing an exception or interrupt. The task stack is accessible from an ISR (i.e., we can push and pop elements onto the task stack when in an ISR), and the interrupt stack is also accessible from a task.

Copyright 2015 Micrium Inc.

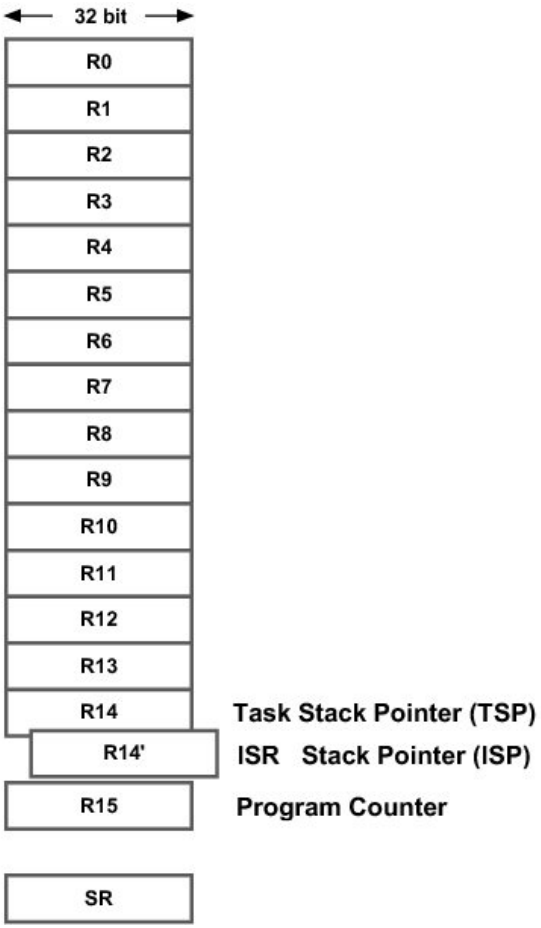


Figure - Fictitious CPU

In µC/OS-III, the stack frame for a ready task is always setup to look as if an interrupt has just occurred and all processor registers were saved onto it. Tasks enter the ready state upon creation and thus their stack frames are pre-initialized by software in a similar manner. Using our fictitious CPU, we'll assume that a stack frame for a task that is ready to be restored is shown in Figure 8-2.

The task stack pointer points to the last register saved onto the task's stack. The program counter (`PC` or `R15`) and status register (`SR`) are the first registers saved onto the stack. In fact, these are saved automatically by the CPU when an exception or interrupt occurs (assuming interrupts are enabled) while the other registers are pushed onto the stack by software in the exception handler. The stack pointer (`SP` or `R14`) is not actually saved on the stack but instead is saved in the task's `OS_TCB`.

The interrupt stack pointer points to the current top-of-stack for the interrupt stack, which is a

Copyright 2015 Micrium Inc.

different memory area. When an ISR executes, the processor uses R14' as the stack pointer for function calls and local arguments.

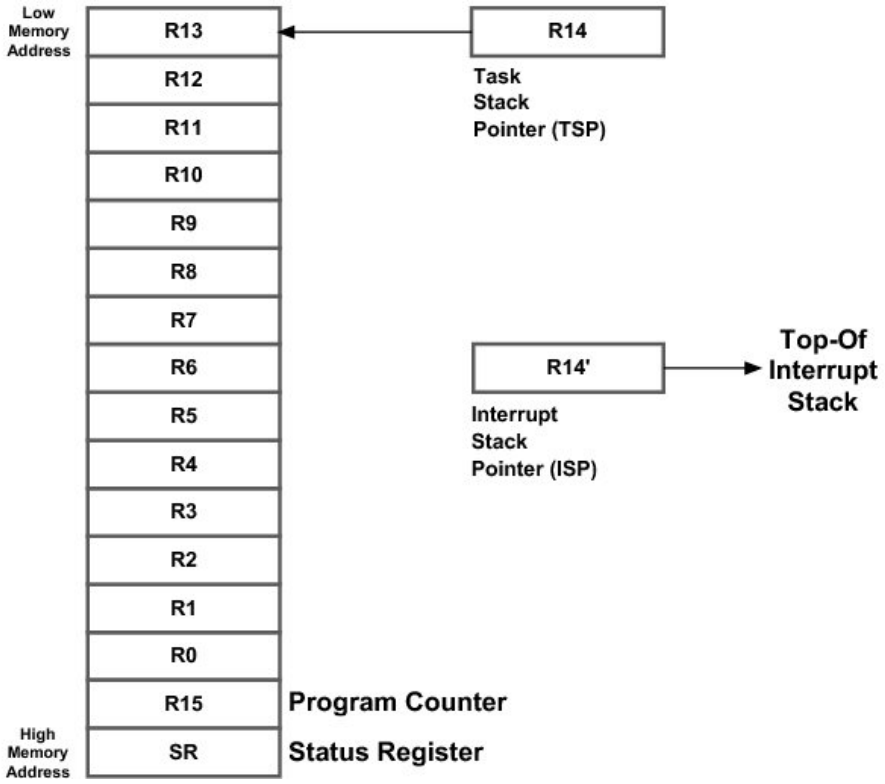


Figure - CPU register stacking order of ready task

There are two types of context switches: one performed from a task and another from an ISR. The task level context switch is implemented by the code in `OSCtxSw()`, which is actually invoked by the macro `OS_TASK_SW()`. A macro is used as there are many ways to invoke `OSCtxSw()` such as software interrupts, trap instructions, or simply calling the function.

The ISR context switch is implemented by `OSIntCtxSw()`. The code for both functions is typically written in assembly language and is found in a file called `os_cpu_a.asm`.

Copyright 2015 Micrium Inc.

OSCtxSw

`OSCtxSw()` (see `os_cpu_a.asm`) is called when the task level scheduler (`OSSched()`) determines that a new high priority task needs to execute. The figure below shows the state of several µC/OS-III variables and data structures just prior to calling `OSCtxSw()`.

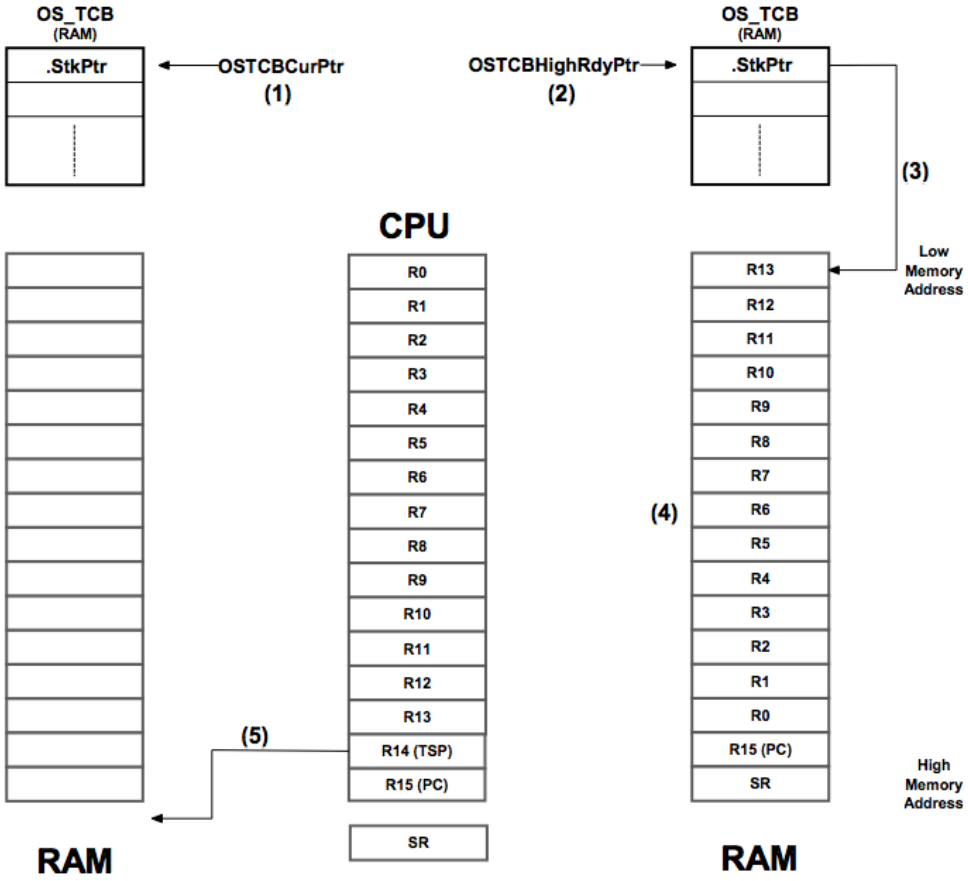


Figure - Variables and data structures prior to calling OSCtxSw()

(1) `OSTCBCurPtr` points to the `OS_TCB` of the task that is currently running and that called `OSSched()`.

(2) `OSSched()` finds the new task to run by having `OSTCBHighRdyPtr` point to its `OS_TCB`.

(3) `OSTCBHighRdyPtr->StkPtr` points to the top of stack of the new task to run.

Copyright 2015 Micrium Inc.

(4) When µC/OS-III creates or suspends a task, it always leaves the stack frame to look as if an interrupt just occurred and all the registers saved onto it. This represents the expected state of the task so it can be resumed.

(5) The CPU's stack pointer points within the stack area (i.e., RAM) of the task that called `OSSched()`. Depending on how `OSCtxSw()` is invoked, the stack pointer may be pointing at the return address of `OSCtxSw()`.

The figure below shows the steps involved in performing the context switch as implemented by `OSCtxSw()`.

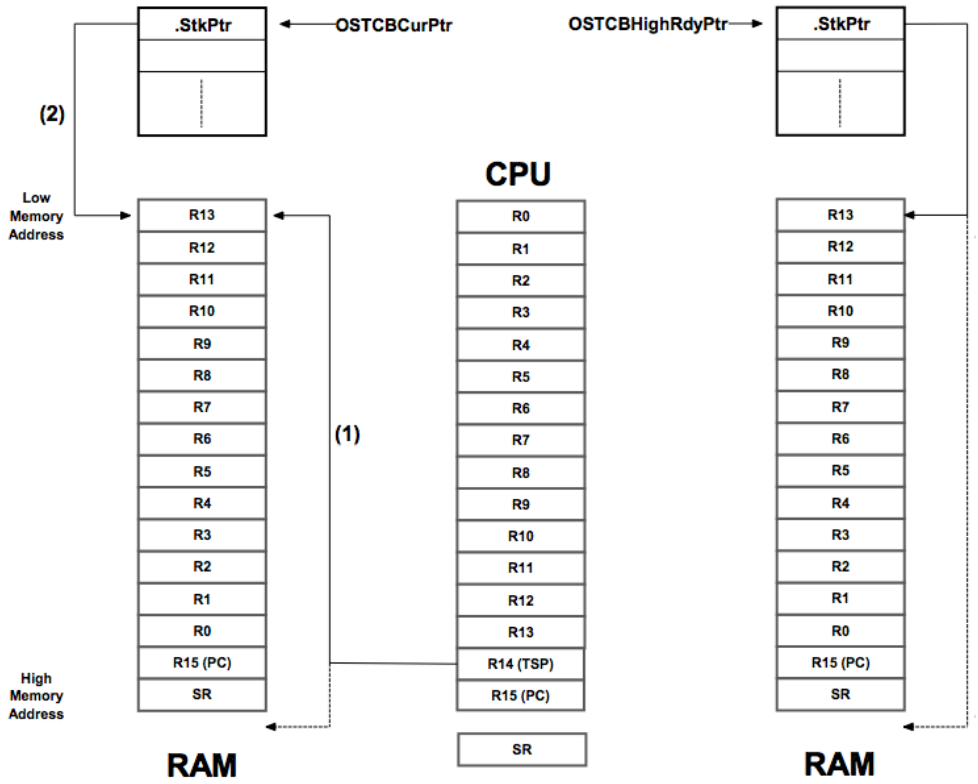


Figure - Operations performed by OSCtxSw()

(1) `OSCtxSw()` begins by saving the status register and program counter of the current task onto the current task's stack. The saving order of register depends on how the CPU

expects the registers on the stack frame when an interrupt occurs. In this case, it is assumed that the SR is stacked first. The remaining registers are then saved onto the stack.

(2) `OSCtxSw()` saves the contents of the CPU's stack pointer into the `OS_TCB` of the task being context switched out. In other words, `OSTCBCurPtr->StkPtr = R14`.

(3) `OSCtxSw()` then loads the CPU stack pointer with the saved top-of-stack from the new task's `OS_TCB`. In other words, `R14 = OSTCBHighRdyPtr->StkPtr`.

(4) Finally, `OSCtxSw()` retrieves the CPU register contents from the new stack. The program counter and status registers are generally retrieved at the same time by executing a return from interrupt instruction.

Copyright 2015 Micrium Inc.

OSIntCtxSw

`OSIntCtxSw()` (see `os_cpu_a.asm`) is called when the ISR level scheduler (`OSIntExit()`) determines that a new high priority task is ready to execute. The figure below shows the state of several µC/OS-III variables and data structures just prior to calling `OSIntCtxSw()`.

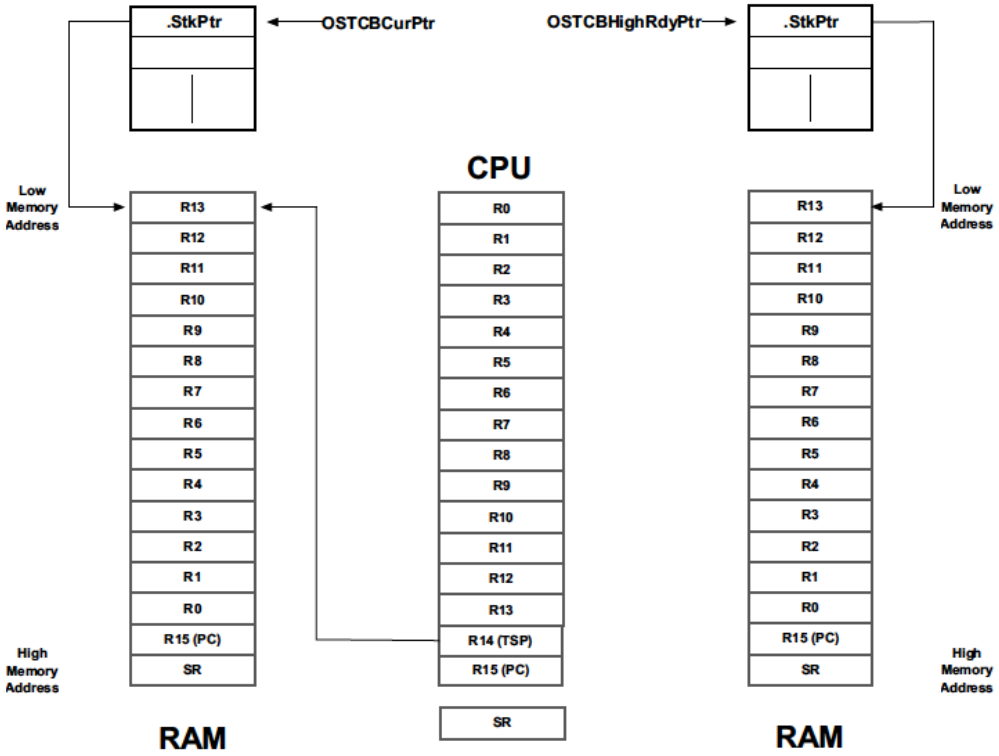


Figure - Variables and data structures prior to calling OSIntCtxSw()

µC/OS-III assumes that CPU registers are saved onto the task's stack at the beginning of an ISR (see Interrupt Management). Because of this, notice that `OSTCBCurPtr->StkPtr` contains a pointer to the top-of-stack pointer of the task being suspended (the one on the left). `OSIntCtxSw()` does not have to worry about saving the CPU registers of the suspended task since that has already been done.

The figure below shows the operations performed by `OSIntCtxSw()` to complete the second half of the context switch. This is exactly the same process as the second half of `OSCtxSw()`.

Copyright 2015 Micrium Inc.

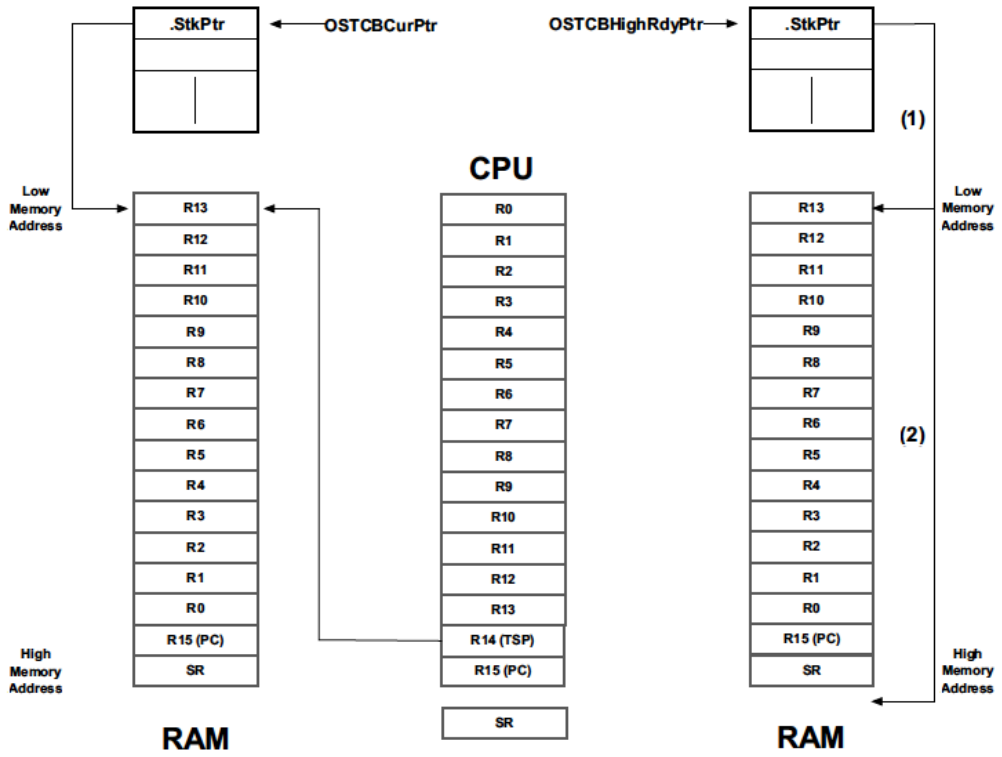


Figure - Operations performed by OSIntCtxSw()

(1) `OSIntCtxSw()` loads the CPU stack pointer with the saved top-of-stack from the new task's `OS_TCB. R14 = OSTCBHighRdyPtr->StkPtr`.

(2) `OSIntCtxSw()` then retrieves the CPU register contents from the new stack. The program counter and status registers are generally retrieved at the same time by executing a return from interrupt instruction.

Copyright 2015 Micrium Inc.

Interrupt Management

An *interrupt* is a hardware mechanism used to inform the CPU that an asynchronous event occurred. When an interrupt is recognized, the CPU saves part (or all) of its context (i.e., registers) and jumps to a special subroutine called an *Interrupt Service Routine* (ISR). The ISR processes the event, and – upon completion of the ISR – the program either returns to the interrupted task, or the highest priority task, if the ISR made a higher priority task ready-to-run.

Interrupts allow a microprocessor to process events when they occur (i.e., asynchronously), which prevents the microprocessor from continuously *polling* (looking at) an event to see if it occurred. Task level response to events is typically better using interrupt mode as opposed to polling mode. Microprocessors allow interrupts to be ignored or recognized through the use of two special instructions: disable interrupts and enable interrupts, respectively.

In a real-time environment, interrupts should be disabled as little as possible. Disabling interrupts affects interrupt latency possibly causing interrupts to be missed.

Processors generally allow interrupts to be nested, which means that while servicing an interrupt, the processor recognizes and services other (more important) interrupts.

One of the most important specifications of a real-time kernel is the maximum amount of time that interrupts are disabled. This is called *interrupt disable time*. All real-time systems disable interrupts to manipulate critical sections of code and re-enable interrupts when critical sections are completed. The longer interrupts are disabled, the higher the interrupt latency.

Interrupt response is defined as the time between the reception of the interrupt and the start of the user code that handles the interrupt. Interrupt response time accounts for the entire overhead involved in handling an interrupt. Typically, the processor’s context (CPU registers) is saved on the stack before the user code is executed.

Interrupt recovery is defined as the time required for the processor to return to the interrupted code or to a higher priority task if the ISR made such a task ready-to-run.

Task latency is defined as the time it takes from the time the interrupt occurs to the time task level code resumes.

Copyright 2015 Micrium Inc.

Handling CPU Interrupts

There are many popular CPU architectures on the market today, and most processors typically handle interrupts from a multitude of sources. For example, a UART receives a character, an Ethernet controller receives a packet, a DMA controller completes a data transfer, an Analog-to-Digital Converter (ADC) completes an analog conversion, a timer expires, *etc*.

In most cases, an *interrupt controller* captures all of the different interrupts presented to the processor as shown in Figure 9-1 (note that the “CPU Interrupt Enable/Disable” is typically part of the CPU, but is shown here separately for sake of the illustration).

Interrupting devices signal the interrupt controller, which then prioritizes the interrupts and presents the highest-priority interrupt to the CPU.

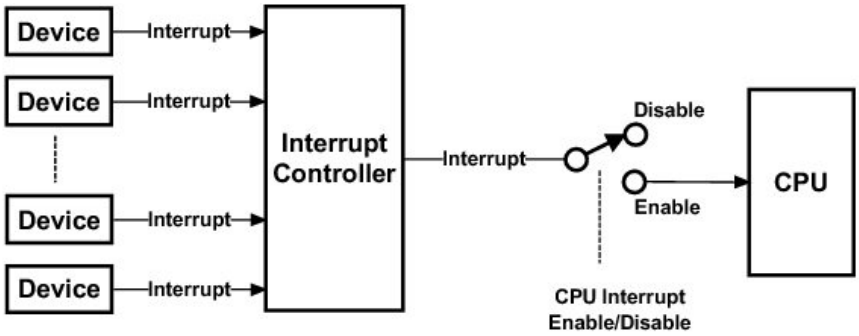


Figure - Interrupt controllers

Modern interrupt controllers have built-in intelligence that enable the user to prioritize interrupts, remember which interrupts are still pending and, in many cases, have the interrupt controller provide the address of the ISR (also called the vector address) directly to the CPU.

If “global” interrupts (i.e., the switch in Figure 9-1) are disabled, the CPU will ignore requests from the interrupt controller, but they will be held pending by the interrupt controller until the CPU re-enables interrupts.

CPUs deal with interrupts using one of two models:

- All interrupts *vector* to a single interrupt handler.
- Each interrupt *vectors* directly to an interrupt handler.

Copyright 2015 Micrium Inc.

Before discussing these two methods, it is important to understand how µC/OS-III handles CPU interrupts.

Typical uC-OS-III Interrupt Service Routine

µC/OS-III requires that interrupt service routines be written in assembly language. However, if a C compiler supports in-line assembly language, the ISR code can be placed directly into a C source file. The pseudo-code for a typical ISR when using µC/OS-III is shown in Listing 9-1.

```
MyISR:                                                          (1)
    Disable all kernel aware interrupts;                        (2)
    Save the CPU registers;                                     (3)
    OSIntNestingCtr++;                                          (4)
    if (OSIntNestingCtr == 1) {                                 (5)
        OSTCBCurPtr->StkPtr = Current task's CPU stack pointer register value;
    }
    Clear interrupting device;                                  (6)
    Re-enable kernel aware interrupts (optional);               (7)
    Call user ISR;                                              (8)
    OSIntExit();                                                (9)
    Restore the CPU registers;                                 (10)
    Return from interrupt;                                     (11)
```

Listing - ISRs under µC/OS-III (assembly language)

(1) As mentioned above, an ISR is typically written in assembly language. `MyISR` corresponds to the name of the handler that will handle the interrupting device.

(2) It is important that all 'kernel aware' interrupts are disabled before going any further. Some processors have interrupts disabled whenever an interrupt handler starts. Others require the user to explicitly disable interrupts as shown here. This step may be tricky if a processor supports different interrupt priority levels. However, there is always a way to solve the problem.

(3) The first thing the interrupt handler must do is save the context of the CPU onto the interrupted task's stack. On some processors, this occurs automatically. However, on most processors it is important to know how to save the CPU registers onto the task's stack. You should save the full "context" of the CPU, which may also include Floating-Point Unit (FPU) registers if the CPU used is equipped with an FPU.

Certain CPUs also automatically switch to a special stack just to process interrupts (i.e., an interrupt stack). This is generally beneficial as it avoids using up valuable task stack space. However, for µC/OS-III, the context of the interrupted task needs to be saved onto that task's stack.

Copyright 2015 Micrium Inc.

If the processor does not have a dedicated stack pointer to handle ISRs then it is possible to implement one in software. Specifically, upon entering the ISR, simply save the current task stack, switch to a dedicated ISR stack, and when done with the ISR switch back to the task stack. Of course, this means that there is additional code to write, however the benefits are enormous since it is not necessary to allocate extra space on the task stacks to accommodate for worst case interrupt stack usage including interrupt nesting.

(4) Next, either call `OSIntEnter()`, or simply increment the variable `OSIntNestingCtr` in assembly language. This is generally quite easy to do and is more efficient than calling `OSIntEnter()`. As its name implies, `OSIntNestingCtr` keeps track of the interrupt nesting level.

(5) If this is the first nested interrupt, you need to save the current value of the stack pointer of the interrupted task into its `OS_TCB`. The global pointer `OSTCBCurPtr` conveniently points to the interrupted task's `OS_TCB`. The very first field in `OS_TCB` is where the stack pointer needs to be saved. In other words, `OSTCBCurPtr->StkPtr` happens to be at offset 0 in the `OS_TCB` (this greatly simplifies assembly language).

(6) At this point, you need to clear the interrupting device so that it does not generate the same interrupt. However, most people defer the clearing of the source and prefer to perform the action within the user ISR handler in "C."

(7) At this point, it is safe to re-enable kernel aware interrupts if you want to support nested interrupts. This step is optional.

(8) At this point, further processing can be deferred to a C function called from assembly language. This is especially useful if there is a large amount of processing to do in the ISR handler. However, as a general rule, keep the ISRs as short as possible. In fact, it is best to simply signal or send a message to a task and let the task handle the details of servicing the interrupting device.

The ISR must call one of the following functions: `OSSemPost()`, `OSTaskSemPost()`, `OSFlagPost()`, `OSQPost()` or `OSTaskQPost()`. This is necessary since the ISR will notify a task, which will service the interrupting device. These are the only functions able to be called from an ISR and they are used to signal or send a message to a task. However, if the ISR does not need to call one of these functions, consider writing the ISR as a "Non Kernel-Aware Interrupt Service Routine," as described in the next section.

Copyright 2015 Micrium Inc.

(9) When the ISR completes, you must call `OSIntExit()` to tell µC/OS-III that the ISR has completed. `OSIntExit()` simply decrements `OSIntNestingCtr` and, if `OSIntNestingCtr` reaches 0, this indicates that the ISR is about to return to task-level code (instead of a previously interrupted ISR). µC/OS-III will need to determine whether there is a higher priority task that needs to run because of one of the nested ISRs. In other words, the ISR might have signaled or sent a message to a higher- priority task waiting for this signal or message. In this case, µC/OS-III will context switch to this higher priority task instead of returning to the interrupted task. In this latter case, `OSIntExit()` does not actually return, but takes a different path.

(10) If the ISR signaled or sent a message to a lower-priority task than the interrupted task, `OSIntExit()` returns. This means that the interrupted task is still the highest-priority task to run and it is important to restore the previously saved registers.

(11) The ISR performs a return from interrupts and so resumes the interrupted task.

NOTE: From this point on, (1) to (6) will be referred to as the *ISR Prologue* and (9) to (11) as the ISR Epilogue.

Copyright 2015 Micrium Inc.

FAST Interrupt Service Routine

The above sequence assumes that the ISR signals or sends a message to a task. However, in many cases, the ISR may not need to notify a task and can simply perform all of its work within the ISR (assuming it can be done quickly). In this case, the ISR will appear as shown in Listing 9-2.

```
MyShortISR:                                                 (1)
            Save enough registers as needed by the ISR;             (2)
            Clear interrupting device;                              (3)
            DO NOT re-enable interrupts;                            (4)
            Call user ISR;                                          (5)
            Restore the saved CPU registers;                        (6)
            Return from interrupt;                                  (7)
```

Listing - Fast ISRs with µC/OS-III

(1) As mentioned above, an ISR is typically written in assembly language. `MyShortISR` corresponds to the name of the handler that will handle the interrupting device.

(2) Here, you save sufficient registers as required to handle the ISR.

(3) The user probably needs to clear the interrupting device to prevent it from generating the same interrupt once the ISR returns.

(4) *Do not* re-enable interrupts at this point since another interrupt could make µC/OS-III calls, forcing a context switch to a higher-priority task. This means that the above ISR would complete, but at a much later time and, possible cause other complications.

(5) Now you can take care of the interrupting device in assembly language or call a C function, if necessary.

(6) Once finished, simply restore the saved CPU registers.

(7) Perform a return from interrupt to resume the interrupted task.

Copyright 2015 Micrium Inc.

Processors with Multiple Interrupt Priorities

There are some processors that actually supports multiple interrupt levels as shown in Figure 9-2.

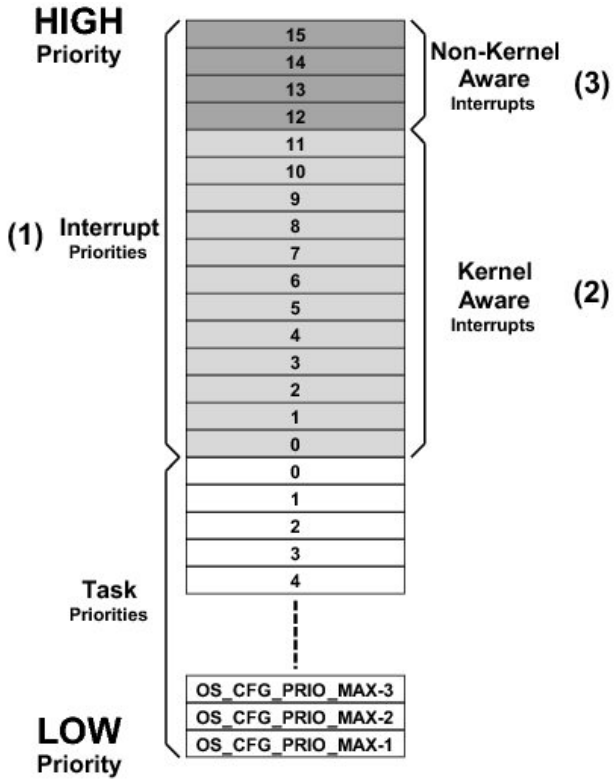


Figure - Kernel Aware and Non-Kernel Aware Interrupts

(1) Here, we are assuming that the processor supports 16 different interrupt priority levels. Priority 0 is the lowest priority while 15 is the highest. As shown, interrupts are always higher in priority than tasks (assuming interrupts are enabled).

(2) The designer of the product decided that interrupt levels 0 through 12 will be allowed to make µC/OS-III 'post' calls to notify tasks that are assigned to service these interrupts. It's important to note that disabling interrupts (when entering critical sections) for task aware interrupts means raising the interrupt mask to level 12 (this is configured in

Copyright 2015 Micrium Inc.

µC/CPU with the `CPU_CFG_KA_IPL_BOUNDARY` configuration constant). In other words, interrupt levels 0 through 11 would be disabled but, levels 12 and above would be allowed.

(3) ISRs at interrupt levels 12 through 15 will **not** be allowed to make any µC/OS-III function calls and are thus implemented as shown in Listing 9-2. It's important to note that since µC/OS-III cannot disable these interrupts, interrupt latency for these interrupts is very short.

Listing 9-3 shows how to implement non-kernel aware ISRs when the processor supports multiple interrupt priorities.

```
MyNonKernelAwareISR:                                        (1)
            Save enough registers as needed by the ISR;             (2)
            Clear interrupting device;                              (3)
            Enable higher priority interrupts (optional);           (4)
            Call user ISR;                                          (5)
            Restore the saved CPU registers;                        (6)
            Return from interrupt;                                  (7)
```

Listing - Non-Kernel Aware ISRs for Processors with Multiple Priority Levels

(1) As mentioned above, an ISR is typically written in assembly language. `MyNonKernelAwareISR` corresponds to the name of the handler that will handle the interrupting device.

(2) Here, you save sufficient registers as required to handle the ISR.

(3) The user probably needs to clear the interrupting device to prevent it from generating the same interrupt once the ISR returns.

(4) At this point, you could enable 'higher priority' interrupts assuming that the CPU disables all interrupts when the interrupt is accepted. However, you **don't** want to enable kernel aware or lower priority interrupts.

(5) Now you can take care of the interrupting device in assembly language or call a C function, if necessary.

Copyright 2015 Micrium Inc.

(6) Once finished, simply restore the saved CPU registers.

(7) Perform a return from interrupt to resume the interrupted task.

Copyright 2015 Micrium Inc.

All Interrupts Vector to a Common Location

Even though an interrupt controller is present in most designs, some CPUs still vector to a common interrupt handler, and the ISR queries the interrupt controller to determine the source of the interrupt. At first glance, this might seem silly since most interrupt controllers are able to force the CPU to jump directly to the proper interrupt handler. It turns out, however, that with µC/OS-III, it's easy to have the interrupt controller vector to a single ISR handler. Listing 9-4 describes the sequence of events to be performed when the interrupt controller forces the CPU to vector to a single location.

```
An interrupt occurs;                                        (1)
 The CPU vectors to a common location;                       (2)
 The ISR code performs the "ISR prologue"                    (3)
 The C handler performs the following:                       (4)
     while (there are still interrupts to process) {         (5)
         Get vector address from interrupt controller;
         Call interrupt handler;
     }
 The "ISR epilogue" is executed;                             (6)
```

Listing - Single interrupt vector for all interrupts

(1) An interrupt occurs from any device. The interrupt controller activates the interrupt pin on the CPU. If there are other interrupts that occur after the first one, the interrupt controller will latch them and properly prioritize the interrupts.

(2) The CPU vectors to a single interrupt handler address. In other words, all interrupts are to be handled by this one interrupt handler.

(3) The ISR executes the "ISR prologue" code needed by µC/OS-III. as previously described. This ensures that all ISRs will be able to make µC/OS-III "post" calls.

(4) You call a µC/OS-III C handler, which will continue processing the ISR. This makes the code easier to write (and read). Notice that interrupts are not re-enabled.

(5) The µC/OS-III C handler then interrogates the interrupt controller and asks it: "Who caused the interrupt?" The interrupt controller will either respond with a number (`0` to `N-1`) or with the address of the interrupt handler for the highest priority interrupting device. Of course, the µC/OS-III C handler will know how to handle the specific interrupt controller since the C handler is written specifically for that controller.

Copyright 2015 Micrium Inc.

If the interrupt controller provides a number between 0 and N-1, the C handler simply uses this number as an index into a table (in ROM or RAM) containing the address of the interrupt service routine associated with the interrupting device. A RAM table is handy to change interrupt handlers at run-time. For many embedded systems, however, the table may also reside in ROM.

If the interrupt controller responds with the address of the interrupt service routine, the C handler only needs to call this function.

In both of the above cases, all interrupt handlers need to be declared as follows:

```
void MyISRHandler (void);
```

There is one such handler for each possible interrupt source (obviously, each having a unique name).

The "`while`" loop terminates when there are no other interrupting devices to service.

(6) The µC/OS-III "ISR epilogue" is executed to see if it is necessary to return to the interrupted task, or switch to a more important one.

A couple of interesting points to note:

- If another device caused an interrupt before the C handler had a chance to query the interrupt controller, most likely the interrupt controller will capture that interrupt. In fact, if that second device happens to be a higher-priority interrupting device, it will most likely be serviced first, as the interrupt controller will prioritize the interrupts.

- The loop will not terminate until all pending interrupts are serviced. This is similar to allowing nested interrupts, but better, since it is not necessary to redo the ISR prologue and epilogue.

The disadvantage of this method is that a high priority interrupt that occurs after the servicing of another interrupt that has already started must wait for that interrupt to complete before it will be serviced. So, the latency of any interrupt, regardless of priority, can be as long as it takes to process the longest interrupt. One more reason to keep your ISRs short.

Copyright 2015 Micrium Inc.

Every Interrupt Vectors to a Unique Location

If the interrupt controller vectors directly to the appropriate interrupt handler, each of the ISRs that are actually used in the application must be written in assembly language as described in Typical uC-OS-III Interrupt Service Routine. This, of course, slightly complicates the design. However, you can copy and paste the majority of the code from one handler to the other and just change what is specific to the actual device.

If the interrupt controller allows the user to query it for the source of the interrupt, it may be possible to simulate the mode in which all interrupts vector to the same location by simply setting all vectors to point to the same location. Most interrupt controllers that vector to a unique location, however, do not allow users to query it for the source of the interrupt since, by definition, having a unique vector for all interrupting devices should not be necessary.

Copyright 2015 Micrium Inc.

Direct and Deferred Post Methods

µC/OS-III handles event posting from interrupts using two different methods: Direct and Deferred Post. The method used in the application is selected by changing the value of `OS_CFG_ISR_POST_DEFERRED_EN` in `os_cfg.h`. When set to `DEF_DISABLED`, µC/OS-III uses the Direct Post Method and when set to `DEF_ENABLED`, µC/OS-III uses the Deferred Post Method. Note that the Deferred Post method is currently DEPRECATED and will be removed in a future µC/OS-III release.

As far as application code and ISRs are concerned, these two methods are completely transparent. It is not necessary to change anything except the configuration value `OS_CFG_ISR_POST_DEFERRED_EN` to switch between the two methods. Of course, changing the configuration constant will require recompiling the product and µC/OS-III.

Before explaining why to use one versus the other, let us review their differences.

Deferred Post Method

In the Deferred Post Method (`OS_CFG_ISR_POST_DEFERRED_EN` is set to `DEF_ENABLED`), instead of disabling interrupts to access critical sections, µC/OS-III locks the scheduler. This avoids having other tasks access critical sections while allowing interrupts to be recognized and serviced. In the Deferred Post Method, interrupts are almost never disabled. The Deferred Post Method is, however, a bit more complex as shown in the figure below. Note that the Deferred Post method is currently DEPRECATED and will be removed in a future µC/OS-III release.

Copyright 2015 Micrium Inc.

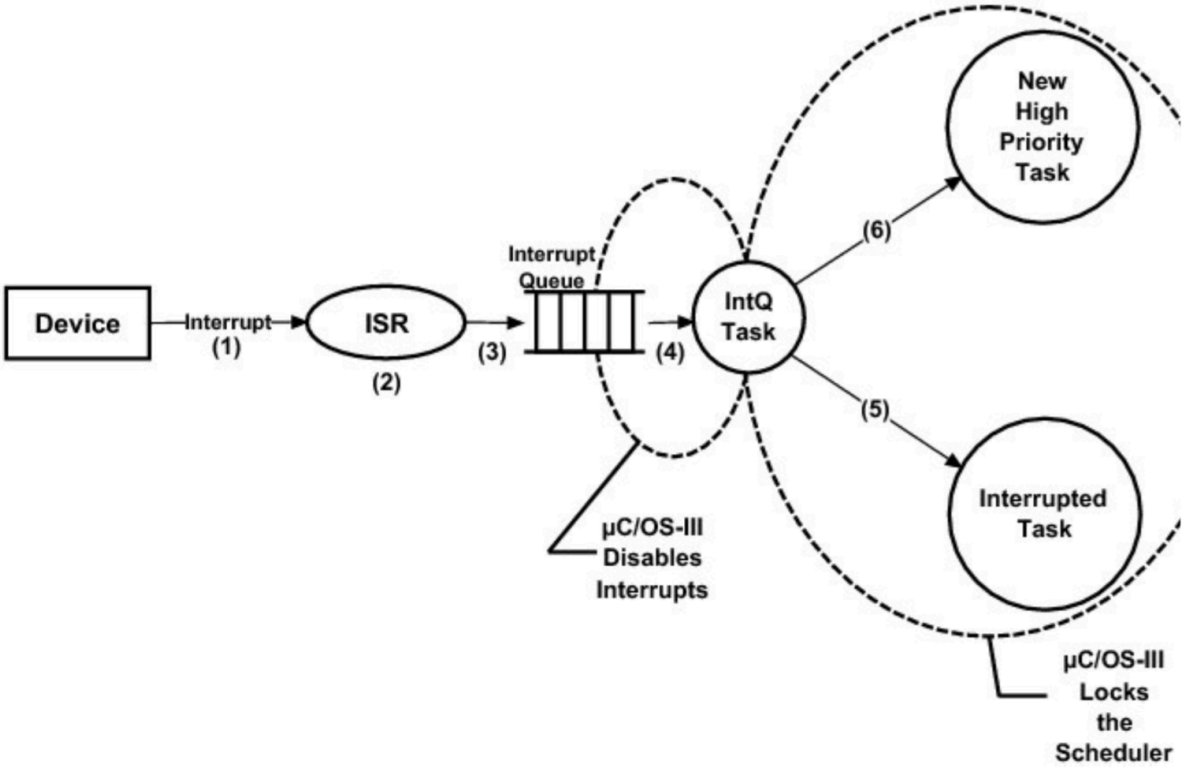


Figure - Deferred Post Method block diagram

(1) A device generates an interrupt.

(2) The ISR responsible for handling the device executes (assuming interrupts are enabled). The device interrupt is the event that a task was waiting for. The task waiting for this interrupt to occur is either higher in priority than the interrupted task, lower, or equal in priority.

(3) The ISR calls one of the post services to signal or send a message to a task. However, instead of performing the post operation, the ISR queues the actual post call along with arguments in a special queue called the *Interrupt Queue*. The ISR then makes the *Interrupt Queue Handler Task* ready-to-run. This task is internal to µC/OS-III and is always the highest priority task (i.e., Priority 0).

(4) At the end of the ISR, µC/OS-III always context switches to the interrupt queue handler task, which then extracts the post command from the queue. We disable interrupts to prevent another interrupt from accessing the interrupt queue while the queue is being

Copyright 2015 Micrium Inc.

emptied. The task then re-enables interrupts, locks the scheduler, and performs the post call as if the post was performed at the task level all along. This effectively manipulates critical sections at the task level.

(5) When the interrupt queue handler task empties the interrupt queue, it makes itself not ready-to-run and then calls the scheduler to determine which task must run next. If the original interrupted task is still the highest priority task, µC/OS-III will resume that task.

(6) If, however, a more important task was made ready-to-run because of the post, µC/OS-III will context switch to that task.

All the extra processing is performed to avoid disabling interrupts during critical sections of code. The extra processing time only consist of copying the post call and arguments into the queue, extracting it back out of the queue, and performing an extra context switch.

Similar to the Direct Post Method, it is easy to determine interrupt latency, interrupt response, interrupt recovery, and task latency, by adding execution times of the pieces of code involved for each as shown below.

Interrupt Latency = Maximum interrupt disable time;

Interrupt Response = Interrupt latency
+ Vectoring to the interrupt handler
+ ISR prologue;

Interrupt Recovery = Handling of the interrupting device
+ Posting a signal or a message to the Interrupt Queue
+ `OSIntExit()`
+ `OSIntCtxSw()` to Interrupt Queue Handler Task;

Task Latency = Interrupt response
+ Interrupt recovery
+ Re-issue the post to the object or task
+ Context switch to task
+ Time scheduler is locked;

The execution times of the µC/OS-III ISR prologue, ISR epilogue, `OSIntExit()`, and

`OSIntCtxSw()`, can be measured independently and should be constant.

It should also be easy to measure the execution time of a post call by using `OS_TS_GET()`. In fact, the post calls should be short in the Deferred Post Method because it only involves copying the post call and its arguments into the interrupt queue.

The difference is that in the Deferred Post Method, interrupts are disabled for a very short amount of time and thus, the first three metrics should be fast. However, task latency is higher as µC/OS-III locks the scheduler to access critical sections.

Direct Post Method

The Direct Post Method is used by µC/OS-II and is replicated in µC/OS-III. This method is enabled by default, the Deferred Post method is currently DEPRECATED and will be removed in a future µC/OS-III release. The figure below shows a task diagram of what takes place in a Direct Post.

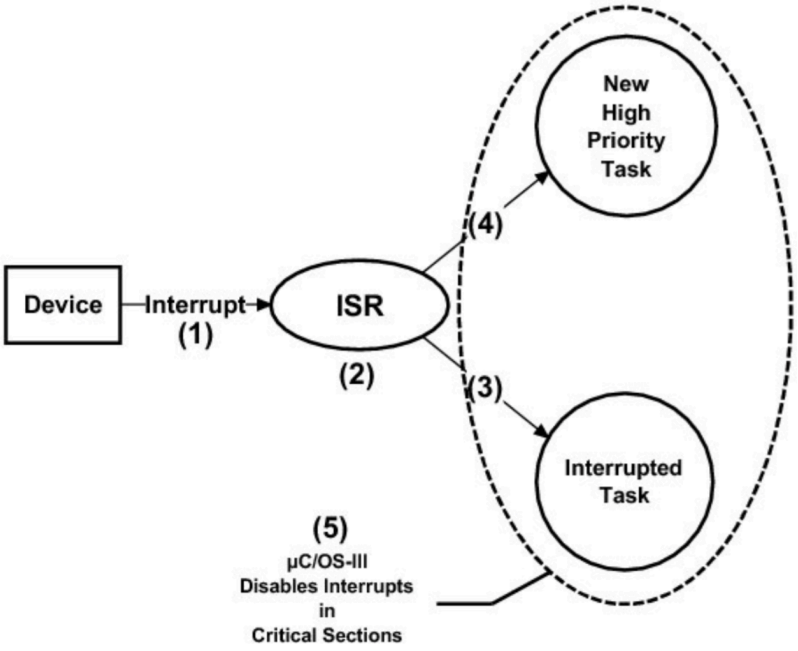


Figure - Direct Post Method

(1) A device generates an interrupt.

(2) The Interrupt Service Routine (ISR) responsible to handle the device executes (assuming interrupts are enabled). The device interrupt is generally the event a task is waiting for. The task waiting for this interrupt to occur either has a higher priority than the interrupted task, or lower (or equal) in priority.

(3) If the ISR made a lower (or equal) priority task ready-to-run then upon completion of the ISR, µC/OS-III returns to the interrupted task exactly at the point the interrupt occurred.

(4) If the ISR made a higher priority task ready-to-run, µC/OS-III will context switch to the new higher-priority task since the more important task was waiting for this device interrupt.

(5) In the Direct Post Method, µC/OS-III must protect critical sections by disabling interrupts as some of these critical sections can be accessed by ISRs.

The above discussion assumed that interrupts were enabled and that the ISR could respond quickly to the interrupting device. However, if the application code makes µC/OS-III service calls (and it will at some point), it is possible that interrupts would be disabled. When `OS_CFG_ISR_POST_DEFERRED_EN` is set to `DEF_DISABLED`, µC/OS-III disables interrupts while accessing critical sections. Thus, interrupts will not be responded to until µC/OS-III re-enables interrupts. Of course, everything was done to keep interrupt disable times as short as possible, but there are complex features of µC/OS-III that disable interrupts for relatively long periods of time.

The key factor in determining whether to use the Direct Post Method is generally the µC/OS-III interrupt disable time. This is fairly easy to determine since the µC/CPU files provided with the µC/OS-III port for the processor used includes code to measure maximum interrupt disable time. This code can be enabled testing purposes and removed when ready to deploy the product. The user would typically not want to leave measurement code in production code to avoid introducing measurement artifacts.

You can determine the interrupt latency, interrupt response, interrupt recovery, and task latency by adding the execution times of the code involved for each, as shown below.

Interrupt Latency = Maximum interrupt disable time;

Copyright 2015 Micrium Inc.

Interrupt Response = Interrupt latency
+ Vectoring to the interrupt handler
+ ISR prologue;

Interrupt Recovery = Handling of the interrupting device
+ Posting a signal or a message to a task
+ `OSIntExit()`
+ `OSIntCtxSw();`

Task Latency = Interrupt response
+ Interrupt recovery
+ Time scheduler is locked;

The execution times of the µC/OS-III ISR prologue, ISR epilogue, `OSIntExit()`, and `OSIntCtxSw()`, can be measured independently and should be fairly constant.

It should also be easy to measure the execution time of a post call by using `OS_TS_GET()`.

In the Direct Post Method, the scheduler is locked only when handling timers and therefore, task latency should be fast if there are few timers with short callbacks expiring at the same time. See Timer Management. µC/OS-III is also able to measure the amount of time the scheduler is locked, providing task latency.

Copyright 2015 Micrium Inc.

Direct vs. Deferred Post Method

In the Direct Post Method, µC/OS-III disables interrupts to access critical sections. In comparison, while in the Deferred Post Method, µC/OS-III locks the scheduler to access the same critical sections.

In the Deferred Post Method, µC/OS-III must still disable interrupts to access the interrupt queue. However, the interrupt disable time is very short and fairly constant. Note that the Deferred Post method is currently DEPRECATED and will be removed in a future µC/OS-III release.

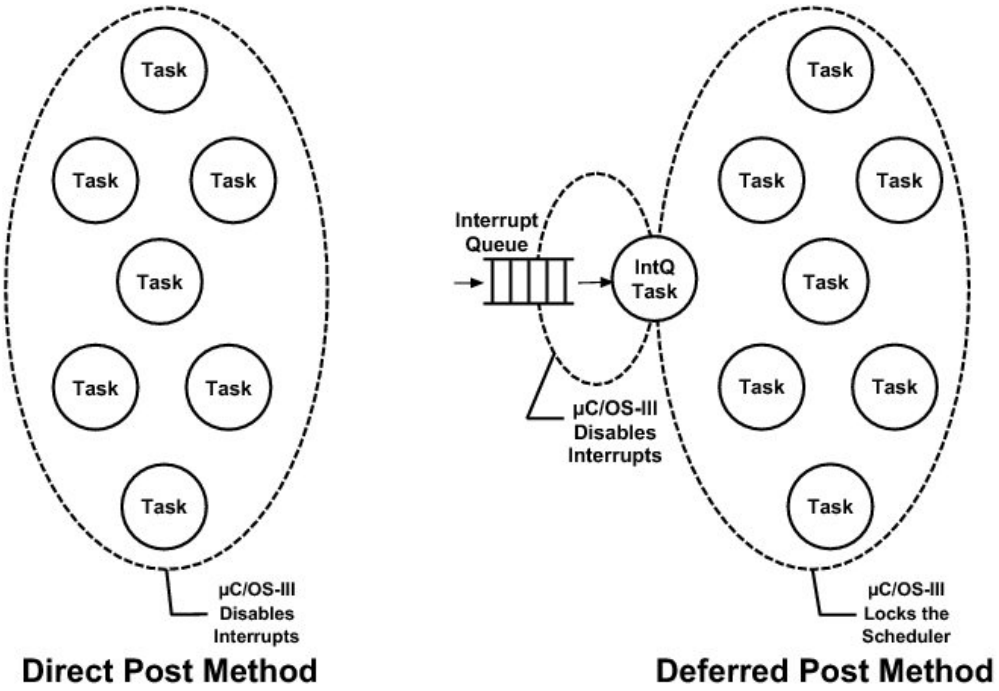


Figure - Direct vs. Deferred Post Methods

If interrupt disable time is critical in the application because there are very fast interrupt sources, and the interrupt disable time of µC/OS-III is not acceptable using the Direct Post Method, use the Deferred Post Method.

Nonetheless, if you are planning on using the features listed in the table below, consider using the Deferred Post Method.

As a general rule, you should always start with the Direct Post Method and see if the performance meets your expectations.

Copyright 2015 Micrium Inc.

Feature	Reason
Multiple tasks at the same priority	Although this is an important feature of µC/OS-III, multiple tasks at the same priority create longer critical sections. However, if there are only a few tasks at the same priority, interrupt latency will be relatively small. If the user does not create multiple tasks at the same priority, the Direct Post Method is recommended.
Event Flags Synchronization	If multiple tasks are waiting on different events, going through all of the tasks waiting for events requires a fair amount of processing time, which means longer critical sections. If only a few tasks (approximately one to five) are waiting on an event flag group, the critical section will be short enough to use the Direct Post Method.
Pend on multiple objects Pending on Multiple Objects	Pending on multiple objects is probably the most complex feature provided by µC/OS-III and requires interrupts to be disabled for fairly long periods of time when using the Direct Post Method. If pending on multiple objects, the Deferred Post Method is highly recommended. If the application does not use this feature, the user may select the Direct Post Method.
Broadcast on Post calls See `OSSemPost()` and `OSQPost()` descriptions.	µC/OS-III disables interrupts while processing a post to multiple tasks in a broadcast. If not using the broadcast option, use the Direct Post Method. Note that broadcasts only apply to semaphores and message queues.

Table - µC/OS-III features to avoid when using the Direct Post Method

Copyright 2015 Micrium Inc.

The Clock Tick

Periodic Tick

µC/OS-III-based systems generally require the presence of a periodic time source called the *clock tick* or *system tick*.

A hardware timer configured to generate an interrupt at a rate between 10 and 1000 Hz provides the clock tick. A tick source may also be obtained by generating an interrupt from an AC power line (typically 50 or 60 Hz). In fact, you can easily derive 100 or 120 Hz by detecting zero crossings of the power line. That being said, if your product is subject to be used in regions that use both power line frequencies then you may need to have the user specify which frequency to use or, have the product automatically detect which region it's in.

The clock tick interrupt can be viewed as the system's heartbeat. The rate is application specific and depends on the desired resolution of this time source. However, the faster the tick rate, the higher the overhead imposed on the system.

The clock tick interrupt allows µC/OS-III to delay tasks for an integral number of clock ticks and provide timeouts when tasks are waiting for events to occur.

The clock tick interrupt must call `OSTimeTick()`. The pseudocode for `OSTimeTick()` is shown in the listing below.

```
void  OSTimeTick (void)
{
    OSTimeTickHook();                                       (1)
#if OS_CFG_ISR_POST_DEFERRED_EN > 0u
    Get timestamp;                                          (2)
    Post “time tick” to the Interrupt Queue;
#else
    Signal the Tick Task;                                   (3)
    Run the round-robin scheduling algorithm;               (4)
    Signal the timer task;                                  (5)
#endif
}
```

Listing - OSTimeTick() pseudocode

(1) The time tick ISR starts by calling a hook function, `OSTimeTickHook()`. The hook

function allows the implementer of the µC/OS-III port to perform additional processing when a tick interrupt occurs. In turn, the tick hook can call a user-defined tick hook if its corresponding pointer, `OS_AppTimeTickHookPtr`, is non-`NULL`. The reason the hook is called first is to give the application immediate access to this periodic time source. This can be useful to read sensors at a regular interval (not as subject to jitter), update Pulse Width Modulation (PWM) registers, and more.

(2) If µC/OS-III is configured for the Deferred Post Method, µC/OS-III reads the current timestamp and defers the call to signal the tick task by placing an appropriate entry in the interrupt queue. The tick task will thus be signaled by the Interrupt Queue Handler Task.

(3) If µC/OS-III is configured for the Direct Post Method, µC/OS-III signals the tick task so that it can process the time delays and timeouts.

(4) µC/OS-III runs the round-robin scheduling algorithm to determine whether the time slot for the current task has expired.

(5) The tick task is also used as the time base for the timers (see Resource Management).

A common misconception is that a system tick is always needed with µC/OS-III. In fact, many low-power applications may not implement the system tick because of the power required to maintain the tick list. In other words, it is not reasonable to continuously power down and power up the product just to maintain the system tick. Since µC/OS-III is a preemptive kernel, an event other than a tick interrupt can wake up a system placed in low power mode by either a keystroke from a keypad or other means. Not having a system tick means that the user is not allowed to use time delays and timeouts on system calls. This is a decision required to be made by the designer of the low-power product.

Copyright 2015 Micrium Inc.

Dynamic Tick

A µC/OS-III-based system using the Dynamic Tick feature needs a programmable timer that can cause and interrupt after a given delay measured in OS ticks. Unlike the periodic tick source, this timer is programmed to interrupt the CPU after a variable amount of time but, needs to be an integral multiple of the tick period. This variable amount of time is called the Tick Step. The Tick Step is determined every time a task needs to delay its execution or to pend on a kernel object with a timeout. The Tick Step is defined as the smallest amount of time the kernel needs to sleep before running the Tick task which is used to wake up the task with the smallest delay or pend timeout. For example, if a task needs to sleep for one second, the Tick task can be run in one second. It does not need to run at a fixed interval until one second has elapsed. However, this assumes that the timer used for the Dynamic tick doesn't overflow during the wait period. For example, you could not set a 16-bit timer to delay for one second if this timer is clocked at 1 MHz because the timer would overflow after 65.536 ms. In this case, you might want to set the tick step to 50 ms and thus load the timer with a count of 50,000. For a delay of 1 second, you would then get 20 Dynamic tick interrupts.

The ISR executed by the Dynamic tick interrupt calls the function `OSTimeDynTick()` instead of `OSTimeTick()`. The Dynamic tick ISR passes the amount OS Ticks that have elapsed since the last Dynamic tick interrupt to the `OSTimeDynTick()` function. By opposition, the `OSTimeTick()` function always assumes that the amount of OS Ticks that have elapsed is one. The `OSTimeDynTick()` function accomplishes the same work as `OSTimeTick()` but uses a variable OS Tick increment. The Tick Task page describes the execution of the Tick task in Periodic and Dynamic modes of operation. Note that to use the Dynamic Tick feature, the µC/OS-III port developer has to implement the Dynamic Tick API described in µC/OS-III Port and `OS_CFG_DYN_TICK_EN` needs to be set to `DEF_ENABLED` in `os_cfg.h`.

Copyright 2015 Micrium Inc.

Pend Lists

A task is placed in a *Pend List* (also called a *Wait List*) when it is waiting on a semaphore to be signaled, a mutual exclusion semaphore to be released, an event flag group to be posted, or a message queue to be posted.

See …	For…	Kernel Object
Resource Management	Semaphores Mutual Exclusion Semaphores	`OS_SEM` `OS_MUTEX`
Synchronization	Semaphores Event Flags	`OS_SEM` `OS_FLAG_GRP`
Message Passing	Message Queues	`OS_Q`

A pend list is similar to the *Ready List*, except that instead of keeping track of tasks that are ready-to-run, the pend list keeps track of tasks waiting for an object to be posted. In addition, the pend list is sorted by priority; the highest priority task waiting on the object is placed at the head of the list, and the lowest priority task waiting on the object is placed at the end of the list.

A pend list is a data structure of type `OS_PEND_LIST`, which consists of three fields as shown in the figure below.

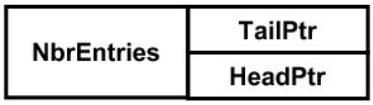


Figure - Pend List

`.NbrEntries`

Contains the current number of entries in the pend list. Each entry in the pend list points to a task that is waiting for the kernel object to be posted.

`.TailPtr`

Is a pointer to the last task in the list (i.e., the lowest priority task).

`.HeadPtr`

Is a pointer to the first task in the list (i.e., the highest priority task).

The figure below indicates that each kernel object using a pend list contains the same three fields at the beginning of the kernel object that we called an `OS_PEND_OBJ`. Notice that the first field is always a “Type” which allows µC/OS-III to know if the kernel object is a semaphore, a mutual exclusion semaphore, an event flag group, or a message queue object.

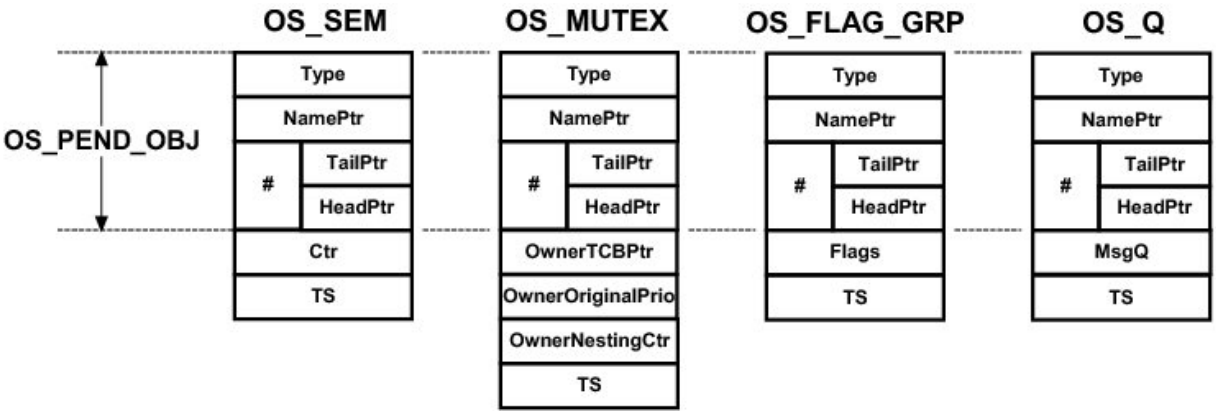


Figure - OS_PEND_OBJ at the beginning of certain kernel objects

The table below shows that the “Type” field of each of the above objects is initialized to contain four ASCII characters when the respective object is created. This allows the user to identify these objects when performing a memory dump using a debugger.

Kernel Object	Type
Semaphore	`‘S’ ‘E’ ‘M’ ‘A’`
Mutual Exclusion Semaphore	`‘M’ ‘U’ ‘T’ ‘X’`
Event Flag Group	`‘F’ ‘L’ ‘A’ ‘G’`
Message Queue	`‘Q’ ‘U’ ‘E’ ‘U’`

Table - Kernel objects with initialized “Type” field

A pend list does not actually point to a task’s `OS_TCB`, but instead points to `OS_PEND_DATA` objects as shown in the figure below. Also, an `OS_PEND_DATA` structure is allocated dynamically on the current task’s stack when a task is placed on a pend list. This implies that a task stack needs to be able to allocate storage for this data structure.

Copyright 2015 Micrium Inc.

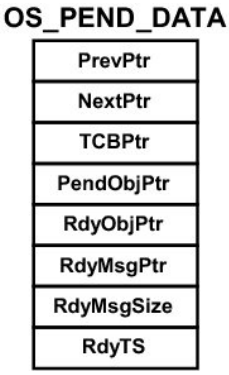


Figure - Pend Data

`.PrevPtr`

Is a pointer to an `OS_PEND_DATA` entry in the pend list. This pointer points to a higher or equal priority task waiting on the kernel object.

`.NextPtr`

Is a pointer to an `OS_PEND_DATA` entry in the pend list. This pointer points to a lower or equal priority task waiting on the kernel object.

`.TCBPtr`

Is a pointer to the `OS_TCB` of the task waiting on the pend list.

`.PendObjPtr`

Is a pointer to the kernel object that the task is pending on. In other words, this pointer can point to an `OS_SEM`, `OS_MUTEX`, `OS_FLAG_GRP` or `OS_Q` by using an `OS_PEND_OBJ` as the common data structure.

`.RdyObjPtr`

Is a pointer to the kernel object that is ready if the task actually waits for multiple kernel objects. See Pending on Multiple Objects for more on this.

`.RdyMsgPtr`

Copyright 2015 Micrium Inc.

Is a pointer to the message posted through `OSQPost()` if the task is pending on multiple kernel objects. Again, see Pending on Multiple Objects.

`.RdyTS`

Is a timestamp of when the kernel object was posted. This is used when a task pends on multiple kernel objects as described in Pending on Multiple Objects.

The figure below shows how all data structures connect to each other when tasks are inserted in a pend list. This drawing assumes that there are two tasks waiting on a semaphore.

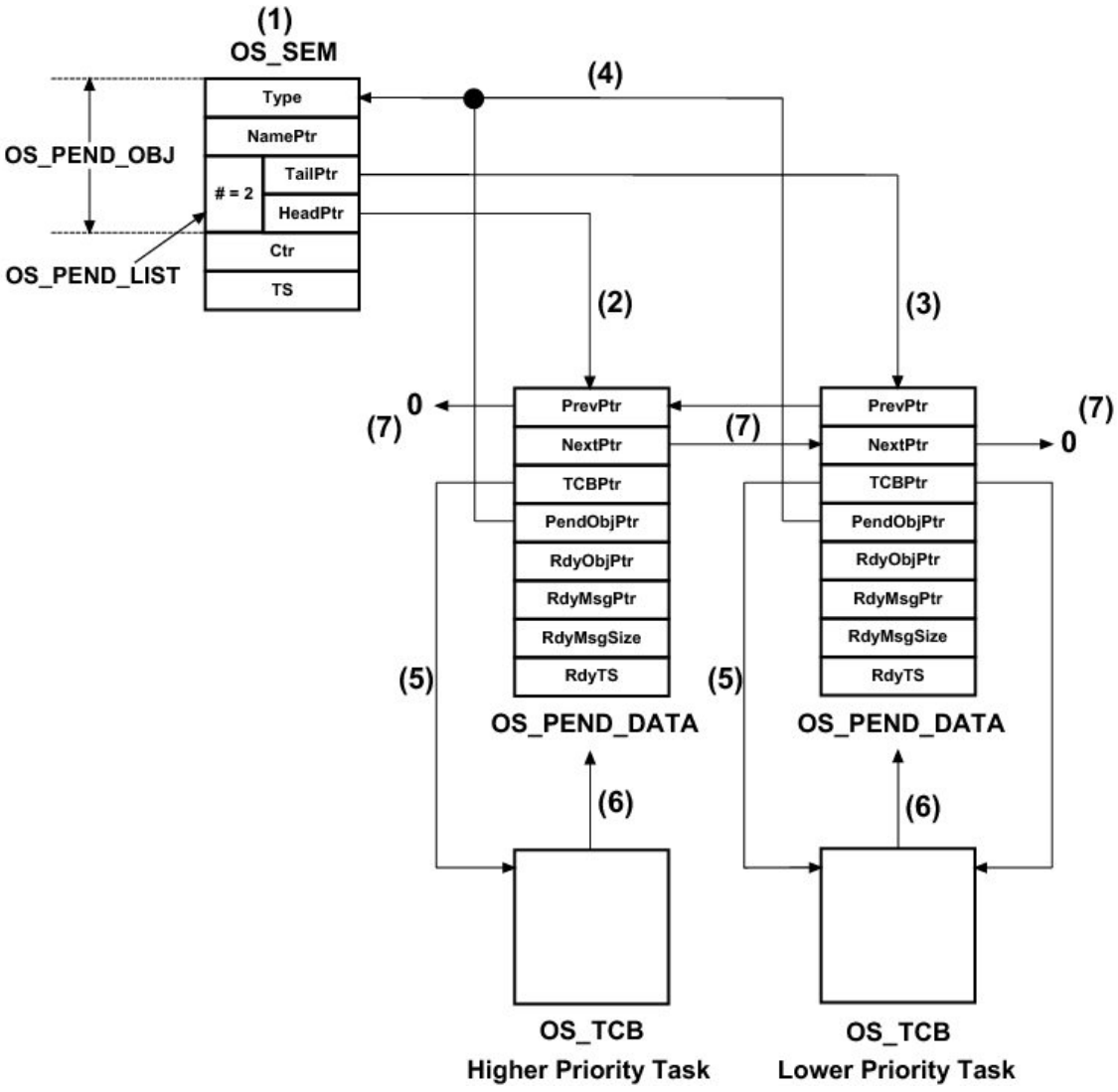


Figure - Pend Data

(1) The `OS_SEM` data type contains an `OS_PEND_OBJ`, which in turn contains an `OS_PEND_LIST`. The `.NbrEntries` field in the pend list indicates that there are two tasks waiting on the semaphore.

(2) The `.HeadPtr` field of the pend list points to the `OS_PEND_DATA` structure associated with the highest priority task waiting on the semaphore.

(3) The `.TailPtr` field of the pend list points to the `OS_PEND_DATA` structure associated with the lowest priority task waiting on the semaphore.

(4) Both `OS_PEND_DATA` structures in turn point back to the `OS_SEM` data structure. The pointers think they are pointing to an `OS_PEND_OBJ`. We know that the `OS_PEND_OBJ` is a semaphore by examining the `.Type` field of the `OS_PEND_OBJ`. `.Type` will contain the four ASCII characters '`S`', '`E`', '`M`' and '`A`'.

(5) Each `OS_PEND_DATA` structure points to its respective `OS_TCB`. In other words, we know which task is pending on the semaphore.

(6) Each task points back to the `OS_PEND_DATA` structure.

(7) Finally, the `OS_PEND_DATA` structure forms a doubly linked list so that the µC/OS-III can easily add or remove entries in this list.

Although this may seem complex, the reasoning will become apparent in Pending on Multiple Objects. For now, you should assume that all of the links are necessary.

The table below shows the functions that µC/OS-III uses to manipulate entries in a pend list. These functions are internal to µC/OS-III and the application code must never call them. The code is found in `os_core.c`.

Copyright 2015 Micrium Inc.

Function	Description
OS_PendListChangePrio()	Change the priority of a task in a pend list
OS_PendListInit()	Initialize a pend list
OS_PendListInsertHead()	Insert an OS_PEND_DATA at the head of the pend list
OS_PendListInsertPrio()	Insert an OS_PEND_DATA in priority order in the pend list
OS_PendListRemove()	Remove multiple OS_PEND_DATA from the pend list
OS_PendListRemove1()	Remove single OS_PEND_DATA from the pend list

Table - Pend List access functions

Copyright 2015 Micrium Inc.

Time Management

µC/OS-III provides time-related services to the application programmer.

In the section Interrupt Management, it was established that µC/OS-III generally requires (as do most kernels) that the user provide a periodic interrupt to keep track of time delays and timeouts. This periodic time source is called a clock tick and should occur between 10 and 1000 times per second, or Hertz (see `OS_CFG_TICK_RATE_HZ` in `os_cfg_app.h`). The actual frequency of the clock tick depends on the desired tick resolution of the application. However, the higher the frequency of the ticker, the higher the overhead.

µC/OS-III provides a number of services to manage time as summarized in the table below, and the code is found in `os_time.c`.

Function Name	Operation
`OSTimeDly()`	Delay execution of a task for “n” ticks
`OSTimeDlyHMSM()`	Delay a task for a user specified time in HH:MM:SS.mmm
`OSTimeDlyResume()`	Resume a delayed task
`OSTimeGet()`	Obtain the current value of the tick counter
`OSTimeSet()`	Set the tick counter to a new value
`OSTimeTick()`	Signal the occurrence of a clock tick

Table - Time Services API summary

The application programmer should refer to µC-OS-III API Reference for a detailed description of these services.

OSTimeDly() – Delay a Task

A task calls this function to suspend execution until some time expires. The calling function will not execute until the specified time expires. This function allows three modes: relative, periodic and absolute.

The listing below shows how to use `OSTimeDly()` in relative mode.

Copyright 2015 Micrium Inc.

```
void  MyTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    while (DEF_ON) {
        :
        :
        OSTimeDly(2,                                (1)
                  OS_OPT_TIME_DLY,                  (2)
                  &err);                            (3)
        /* Check “err” */                           (4)
        :
        :
    }
}
```

Listing - OSTimeDly() - Relative

(1) The first argument specifies the amount of time delay (in number of ticks) from when the function is called. For the example above, if the tick rate (`OS_CFG_TICK_RATE_HZ` in `os_cfg_app.h`) is set to 1000 Hz, the user is asking to suspend the current task for approximately 2 milliseconds. However, the value is not accurate since the count starts from the next tick which could occur almost immediately. This will be explained shortly.

(2) Specifying `OS_OPT_TIME_DLY` indicates that the user wants to use “relative” mode.

(3) As with most µC/OS-III services an error return value will be returned. The example should return `OS_ERR_NONE` because the arguments are all valid.

(4) You should always check the error code returned by µC/OS-III. If “`err`” does not contain `OS_ERR_NONE`, `OSTimeDly()` did not perform the intended work. For example, another task could remove the time delay suspension by calling `OSTimeDlyResume()` and when `MyTask()` returns, it would not have returned because the time had expired.

As mentioned above, the delay is not accurate. Refer to the figure below and its description below to understand why.

Copyright 2015 Micrium Inc.

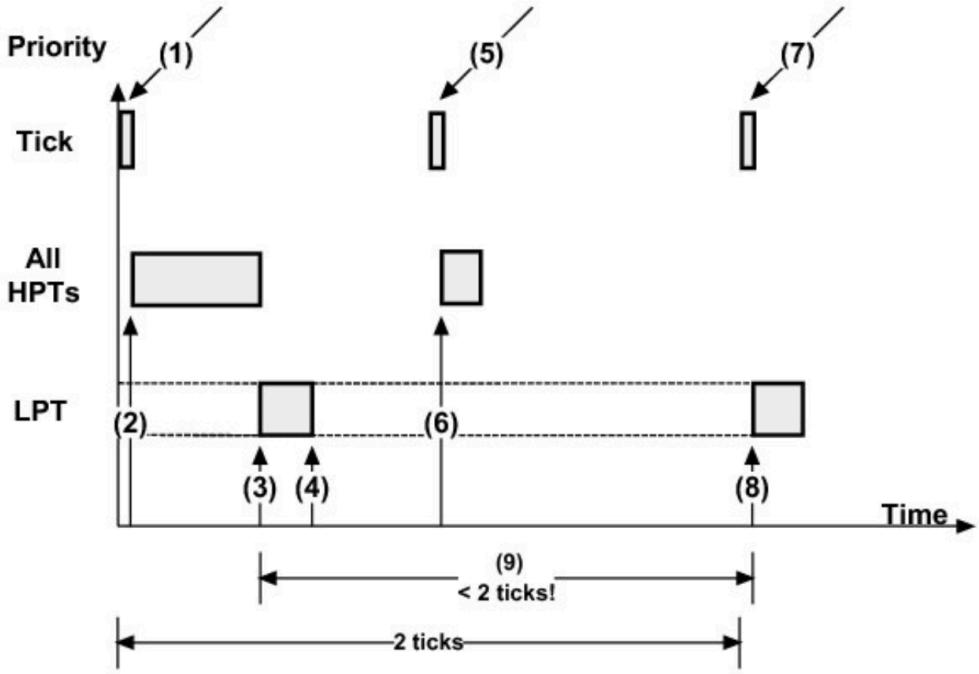


Figure - OSTimeDly() - Relative

(1) We get a tick interrupt and µC/OS-III services the ISR.

(2) At the end of the ISR, all Higher Priority Tasks (HPTs) execute. The execution time of HPTs is unknown and can vary.

(3) Once all HPTs have executed, µC/OS-III runs the task that has called `OSTimeDly()` as shown in the listing above. For the sake of discussion, it is assumed that this task is a lower priority task (LPT).

(4) The task calls `OSTimeDly()` and specifies to delay for two ticks in “relative” mode. At this point, µC/OS-III places the current task in the tick list where it will wait for two ticks to expire. The delayed task consumes zero CPU time while waiting for the time to expire.

(5) The next tick occurs. If there are HPTs waiting for this particular tick, µC/OS-III will schedule them to run at the end of the ISR.

(6) The HPTs execute.

Copyright 2015 Micrium Inc.

(7) The next tick interrupt occurs. This is the tick that the LPT was waiting for and will now be made ready-to-run by µC/OS-III.

(8) Since there are no HPTs to execute on this tick, µC/OS-III switches to the LPT.

(9) Given the execution time of the HPTs, the time delay is not exactly two ticks, as requested. In fact, it is virtually impossible to obtain a delay of exactly the desired number of ticks. You might ask for a delay of two ticks, but the very next tick could occur almost immediately after calling `OSTimeDly()`! Just imagine what might happen if all HPTs took longer to execute and pushed 3 and 4 further to the right. In this case, the delay would actually appear as one tick instead of two.

`OSTimeDly()` can also be called with the `OS_OPT_TIME_PERIODIC` option as shown in the listing below . This option allows delaying the task until the tick counter reaches a certain periodic match value and thus ensures that the spacing in time is always the same as it is not subject to CPU load variations.

µC/OS-III determines the “match value” of `OSTickCtr` to determine when the task will need to wake up based on the desired period. This is shown in the figure below. µC/OS-III checks to ensure that if the match is computed such that it represents a value that has already gone by then, the delay will be zero.

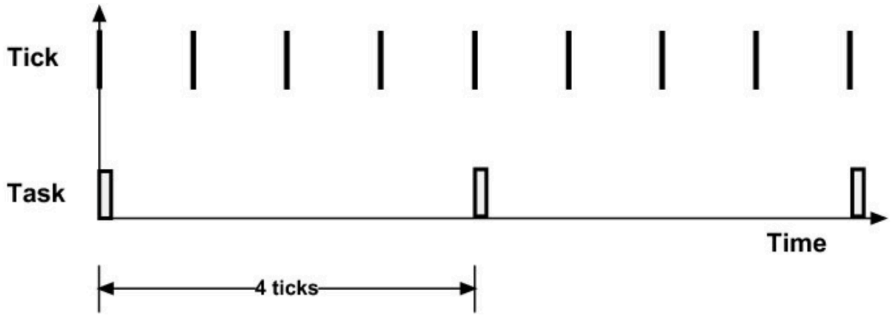


Figure - OSTimeDly() - Periodic

```
 {
    OS_ERR   err;
    :
    :
    while (DEF_ON) {
        OSTimeDly(4,                                (1)
                  OS_OPT_TIME_PERIODIC,             (2)
                  &err);
        /* Check “err” */                           (3)
        :
        :
    }
}
```

Listing - OSTimeDly() - Periodic

(1) The first argument specifies the period for the task to execute, specifically every four ticks. Of course, if the task is a low-priority task, µC/OS-III only schedules and runs the task based on its priority relative to what else needs to be executed.

(2) Specifying `OS_OPT_TIME_PERIODIC` indicates that the task is to be ready-to-run when the tick counter reaches the desired period from the previous call.

(3) You should always check the error code returned by µC/OS-III.

Relative and Periodic modes might not look different, but they are. In Relative mode, it is possible to miss one of the ticks when the system is heavily loaded, missing a tick or more on occasion. In Periodic mode, the task may still execute later, but it will always be synchronized to the desired number of ticks. In fact, Periodic mode is the preferred mode to use to implement a time-of-day clock.

Finally, you can use the absolute mode to perform a specific action at a fixed time after power up. For example, turn off a light 10 seconds after the product powers up. In this case, you would specify `OS_OPT_TIME_MATCH` while “`dly`” actually corresponds to the desired value of `OSTickCtr` you want to reach. However, you should use the `OS_OPT_TIME_MATCH` with care because somewhere else in your code you can change the value of `OSTickCtr` by calling `OSTimeSet()` as described in Time Management.

To summarize, the task will wake up when `OSTickCtr` reaches the following value:

Value of “opt”	Task wakes up when

OS_OPT_TIME_DLY	OSTickCtr + dly
OS_OPT_TIME_PERIODIC	OSTCBCurPtr->TickCtrPrev + dly
OS_OPT_TIME_MATCH	dly

OSTimeDlyHMSM() – Delay a Task for a Specified Time

If enabled, a task may call this function to suspend execution until some time expires by specifying the length of time in a more user-friendly way. Specifically, you can specify the delay in hours, minutes, seconds, and milliseconds (thus the HMSM). This function only works in “Relative” mode.

The listing below indicates how to use OSTimeDlyHMSM().

```
 void  MyTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    while (DEF_ON) {
        :
        :
        OSTimeDlyHMSM(0,                              (1)
                      0,
                      1,
                      0,
                      OS_OPT_TIME_HMSM_STRICT,        (2)
                      &err);                          (3)
        /* Check “err” */
        :
        :
    }
}
```

Listing - OSTimeDlyHMSM()

(1) The first four arguments specify the amount of time delay (in hours, minutes, seconds, and milliseconds) from this point in time. In the above example, the task should delay for 1 second. The resolution greatly depends on the tick rate. For example, if the tick rate (OS_CFG_TICK_RATE_HZ in os_cfg_app.h) is set to 1000 Hz there is technically a resolution of 1 millisecond. If the tick rate is 100 Hz then the delay of the current task is in increments of 10 milliseconds. Again, given the relative nature of this call, the actual delay may not be accurate.

(2) Specifying OS_OPT_TIME_HMSM_STRICT verifies that the user strictly passes valid values for

hours, minutes, seconds and milliseconds. Valid hours are 0 to 99, valid minutes are 0 to 59, valid seconds are 0 to 59, and valid milliseconds are 0 to 999.

If specifying `OS_OPT_TIME_HMSM_NON_STRICT`, the function will accept nearly any value for hours (between 0 to 999), minutes (from 0 to 9999), seconds (any value, up to `65,535`), and milliseconds (any value, up to `4,294,967,295`). `OSTimeDlyHMSM(203, 101, 69, 10000)` may be accepted. Whether or not this makes sense is a different story.

The reason hours is limited to 999 is that time delays typically use 32-bit values to keep track of ticks. If the tick rate is set at 1000 Hz then, it is possible to only track 4,294,967 seconds, which corresponds to 1,193 hours, and therefore 999 is a reasonable limit.

(3) As with most µC/OS-III services the user will receive an error return value. The example should return `OS_ERR_NONE` since the arguments in the listing are all valid. Refer to µC-OS-III API Reference for a list of possible error codes.

Even though µC/OS-III allows for very long delays for tasks, it is actually not recommended to delay tasks for a long time. The reason is that there is no indication that the task is actually "alive" unless it is possible to monitor the amount of time remaining for the delay. It is better to have the task wake up approximately every minute or so, and have it "tell you" that it is still ok.

`OSTimeDly()` is often used to create periodic tasks (tasks that execute periodically). For example, it is possible to have a task that scans a keyboard every 50 milliseconds and another task that reads analog inputs every 10 milliseconds, etc.

OSTimeDlyResume() – Resume a Delayed Task

A task can resume another task that called `OSTimeDly()` or `OSTimeDlyHMSM()` by calling `OSTimeDlyResume()`. The listing below shows how to use `OSTimeDlyResume()`. The task that delayed itself will not know that it was resumed, but will think that the delay expired. Because of this, use this function with great care.

Copyright 2015 Micrium Inc.

```
OS_TCB  MyTaskTCB;


void  MyTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    while (DEF_ON) {
        :
        :
        OSTimeDly(10,
                  OS_OPT_TIME_DLY,
                  &err);
        /* Check “err” */
        :
        :
    }
}


void  MyOtherTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    while (DEF_ON) {
        :
        :
        OSTimeDlyResume(&MyTaskTCB,
                        &err);
        /* Check “err” */
        :
        :
    }
}
```

Listing - OSTimeDlyResume()

OSTimeGet() – Get Value of the Tick Counter

µC/OS-III increments a tick counter every time a tick interrupt occurs. This counter allows the application to make coarse time measurements and have some notion of time (after power up).

`OSTimeGet()` allows the user to take a snapshot of the tick counter. You can use this value to delay a task for a specific number of ticks and repeat this periodically without losing track of time.

`OSTimeSet()` allows the user to change the current value of the tick counter. Although µC/OS-III allows for this, it is recommended to use this function with great care especially if you use ‘match’ time delays.

Copyright 2015 Micrium Inc.

OSTimeSet() and OSTimeGet()

µC/OS-III increments a tick counter every time a tick interrupt occurs. This counter allows the application to make coarse time measurements and have some notion of time (after power up).

`OSTimeGet()` allows the user to take a snapshot of the tick counter. You can use this value to delay a task for a specific number of ticks and repeat this periodically without losing track of time.

`OSTimeSet()` allows the user to change the current value of the tick counter. Although µC/OS-III allows for this, it is recommended to use this function with great care especially if you use ‘match’ time delays.

OSTimeTick() – Signal a Clock Tick

The tick Interrupt Service Routine (ISR) must call this function every time a tick interrupt occurs. C/OS-III uses this function to update time delays and timeouts used by other system calls. OSTimeTick() is considered an internal function to C/OS-III.

Copyright 2015 Micrium Inc.

Timer Management

Timer Services

µC/OS-III provides timer services to the application programmer and code to handle timers is found in `os_tmr.c`. Timer services are enabled when setting `OS_CFG_TMR_EN` to `DEF_ENABLED` in `os_cfg.h`.

Timers are down counters that perform an *action* when the counter reaches zero. The user provides the action through a *callback* function (or simply *callback*). A callback is a user-declared function that will be called when the timer expires. The callback can be used to turn a light on or off, start a motor, or perform other actions. However, it is important to never make blocking calls within a callback function (i.e., call `OSTimeDly()`, `OSTimeDlyHMSM()`, `OS???Pend()`, or anything that causes the timer task to block or be deleted).

Timers are useful in protocol stacks (retransmission timers, for example), and can also be used to poll I/O devices at predefined intervals.

An application can have any number of timers (limited only by the amount of RAM available). Timer services (i.e. functions) in µC/OS-III start with the `OSTmr???()` prefix, and the services available to the application programmer are described in µC-OS-III API Reference.

The resolution of all the timers managed by µC/OS-III is determined by the configuration constant: `OS_CFG_TMR_TASK_RATE_HZ`, which is expressed in Hertz (Hz). So, if the timer task (described later) rate is set to 10, all timers have a resolution of 1/10th of a second (ticks in the diagrams to follow). In fact, this is the typical recommended value for the timer task. Timers are to be used with “coarse” granularity.

µC/OS-III provides a number of services to manage timers as summarized in the table below.

Copyright 2015 Micrium Inc.

Function Name	Operation
OSTmrCreate()	Create and specify the operating mode of the timer.
OSTmrDel()	Delete a timer.
OSTmrRemainGet()	Obtain the remaining time left before the timer expires.
OSTmrStart()	Start (or restart) a timer.
OSTmrStateGet()	Obtain the current state of a timer.
OSTmrStop()	Stop the countdown process of a timer.

Table - Timer API summary

A timer needs to be created before it can be used. You create a timer by calling OSTmrCreate() and specify a number of arguments to this function based on how the timer is to operate. Once the timer operation is specified, its operating mode cannot be changed unless the timer is deleted and recreated. The function prototype for OSTmrCreate() is shown below as a quick reference:

```
void  OSTmrCreate (OS_TMR              *p_tmr,           /* Pointer to timer     */
                   CPU_CHAR            *p_name,          /* Name of timer, ASCII */
                   OS_TICK              dly,             /* Initial delay        */
                   OS_TICK              period,          /* Repeat period        */
                   OS_OPT               opt,             /* Options              */
                   OS_TMR_CALLBACK_PTR  p_callback,      /* Fnct to call at 0    */
                   void                *p_callback_arg,  /* Arg. to callback     */
                   OS_ERR              *p_err)
```

Once created, a timer can be started (or restarted) and stopped as often as is necessary. Timers can be created to operate in one of three modes: One-shot, Periodic (no initial delay), and Periodic (with initial delay).

One-Shot Timers

As its name implies, a one-shot timer will countdown from its initial value, call the callback function when it reaches zero, and stop. The figure below shows a timing diagram of this operation. The countdown is initiated by calling OSTmrStart(). At the completion of the time delay, the callback function is called, assuming a callback function was provided when the timer was created. Once completed, the timer does not do anything unless restarted by calling OSTmrStart(), at which point the process starts over.

You terminate the countdown process of a timer (before it reaches zero) by calling OSTmrStop(). In this case, you can specify that the callback function be called or not.

Copyright 2015 Micrium Inc.

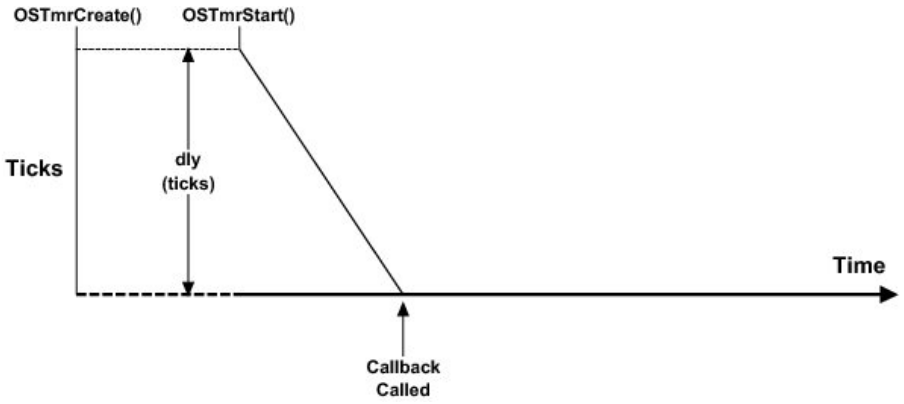


Figure - One Shot Timers (dly > 0, period == 0)

As shown in the figure below, a one-shot timer can be retriggered by calling `OSTmrStart()` before the timer reaches zero. This feature can be used to implement watchdogs and similar safeguards.

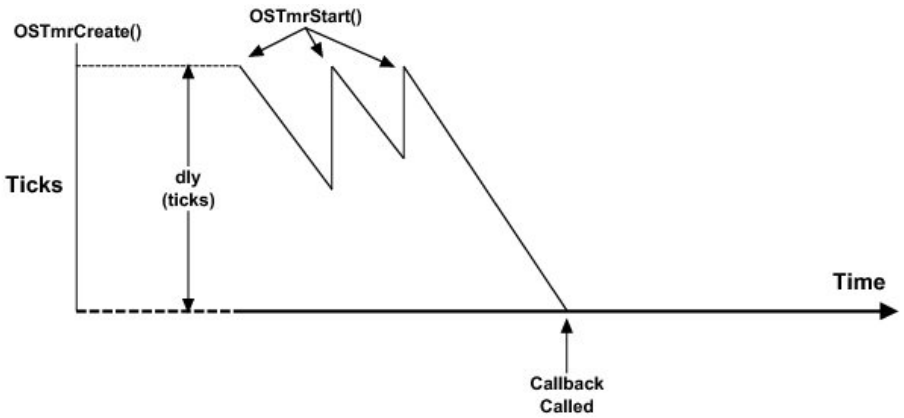


Figure - Retriggering a One Shot Timer

Periodic (no initial delay)

As indicated in the figure below, timers can be configured for periodic mode. When the countdown expires, the callback function is called, the timer is automatically reloaded, and the process is repeated. If specifying a delay of zero (i.e., `dly == 0`) when the timer is created and, when started, the timer immediately uses the "`period`" as the reload value. You can call `OSTmrStart()` at any point in the countdown to restart the process.

Copyright 2015 Micrium Inc.

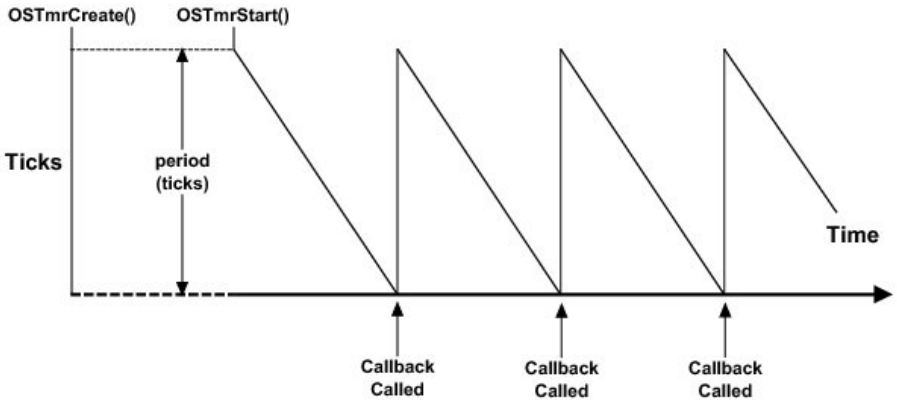


Figure - Periodic Timers (dly == 0, period > 0)

Periodic (with initial delay)

As shown in the figure below, timers can be configured for periodic mode with an initial delay that is different than its period. The first countdown count comes from the "`dly`" argument passed in the `OSTmrCreate()` call, and the reload value is the "`period`". You can call `OSTmrStart()` to restart the process including the initial delay.

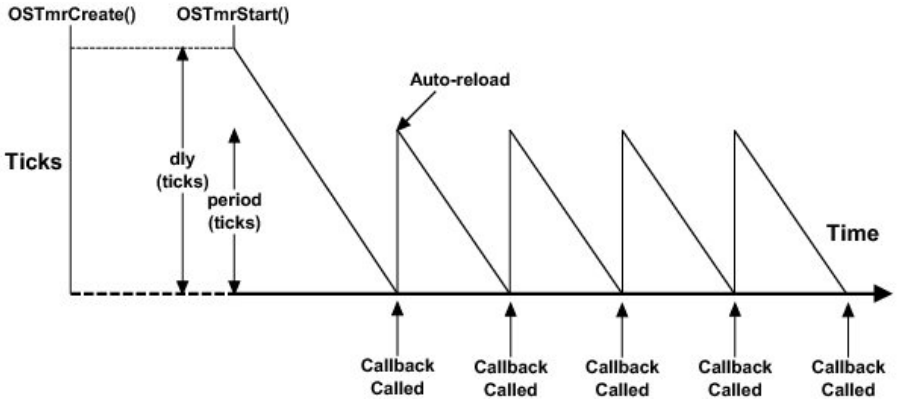


Figure - Periodic Timers (dly > 0, period > 0)

Timer Management Internals

Timer States

The figure below shows the state diagram of a timer.

Tasks can call `OSTmrStateGet()` to find out the state of a timer. Also, at any time during the countdown process, the application code can call `OSTmrRemainGet()` to find out how much time remains before the timer reaches zero (0). The value returned is expressed in “timer ticks.” If timers are decremented at a rate of 10 Hz then a count of 50 corresponds to 5 seconds. If the timer is in the stop state, the time remaining will correspond to either the initial delay (one shot or periodic with initial delay), or the period if the timer is configured for periodic without initial delay.

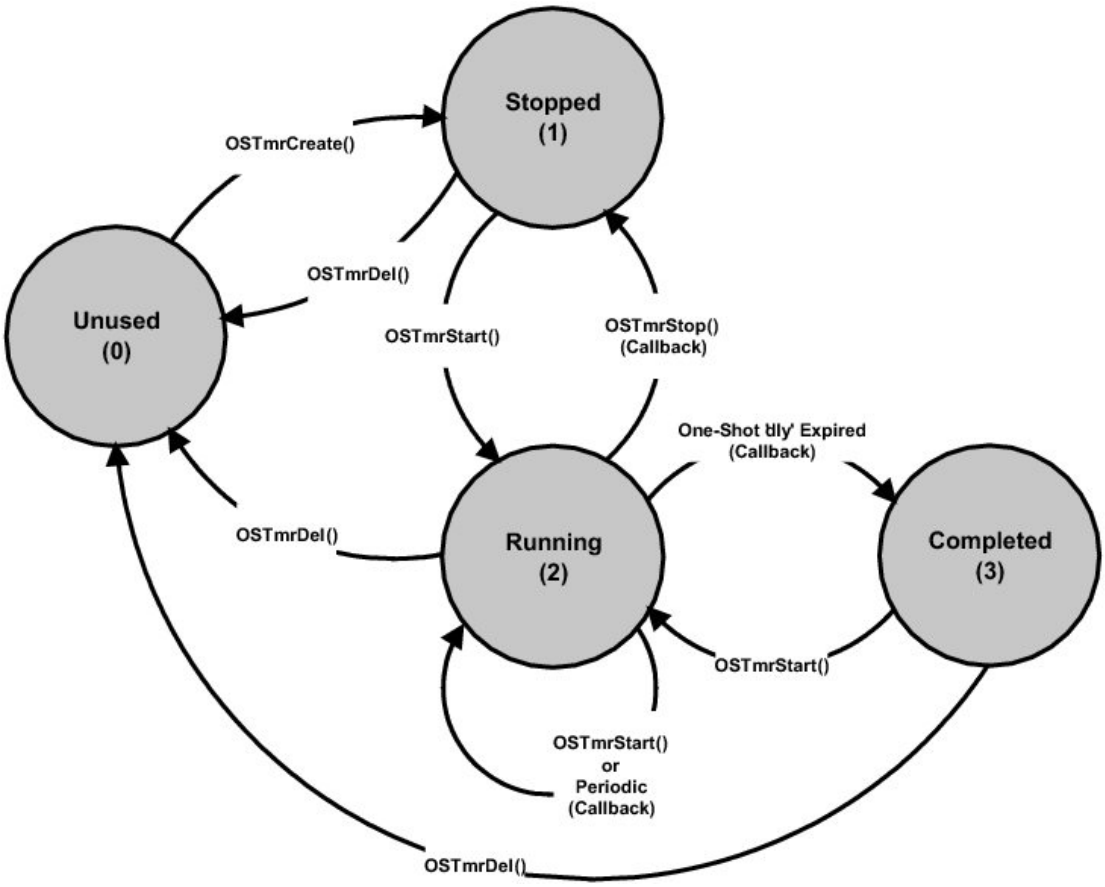


Figure - Timer State Diagram

(1) The “Unused” state is a timer that has not been created or has been “deleted.” In other words, µC/OS-III does not know about this timer.

(2) When creating a timer or calling `OSTmrStop()`, the timer is placed in the “stopped” state.

(3) A timer is placed in running state when calling `OSTmrStart()`. The timer stays in that state unless it’s stopped, deleted, or completes its one shot.

(4) The “Completed” state is the state a one-shot timer is in when its delay expires.

OS_TMR

A timer is a kernel object as defined by the `OS_TMR` data type (see `os.h`) as shown in the listing below:

The services provided by µC/OS-III to manage timers are implemented in the file `os_tmr.c`. Timer services are enabled at compile time by setting the configuration constant `OS_CFG_TMR_EN` to `DEF_ENABLED` in `os_cfg.h`.

```
typedef  struct  os_tmr  OS_TMR;              (1)


struct  os_tmr {
    OS_OBJ_TYPE          Type;                (2)
    CPU_CHAR            *NamePtr;             (3)
    OS_TMR_CALLBACK_PTR  CallbackPtr;         (4)
    void                *CallbackPtrArg;      (5)
    OS_TMR              *NextPtr;             (6)
    OS_TMR              *PrevPtr;
    OS_TICK              Remain;              (7)
    OS_TICK              Dly;                 (8)
    OS_TICK              Period;              (9)
    OS_OPT               Opt;                (10)
    OS_STATE             State;              (11)
};
```

Listing - OS_TMR data type

(1) In µC/OS-III, all structures are given a data type. In fact, all data types start with “`OS_`” and are all uppercase. When a timer is declared, you simply use `OS_TMR` as the data type

of the variable used to declare the timer.

(2) The structure starts with a "Type" field, which allows it to be recognized by µC/OS-III as a timer. Other kernel objects will also have a "Type" as the first member of the structure. If a function is passed a kernel object, µC/OS-III is able to confirm that it is passed the proper data type (assuming OS_CFG_OBJ_TYPE_CHK_EN is set to DEF_ENABLED in os_cfg.h). For example, if passing a message queue (OS_Q) to a timer service (for example OSTmrStart()) then µC/OS-III will be able to recognize that an invalid object was passed, and return an error code accordingly.

(3) Each kernel object can be given a name for easier recognition by debuggers or µC/Probe. This member is simply a pointer to an ASCII string which is assumed to be NUL terminated.

(4) The .CallbackPtr member is a pointer to a function that is called when the timer expires. If a timer is created and passed a NULL pointer, a callback would not be called when the timer expires.

(5) If there is a non-NULL .CallbackPtr then the application code could have also specified that the callback be called with an argument when the timer expires. This is the argument that would be passed in this call.

(6) .NextPtr and .PrevPtr are pointers used to link a timer in a doubly linked list. These are described later.

(7) The .Remain field contains the amount of time remaining for the timer to expire. This value is updated each time the timer task executes (described later). The value is expressed in multiples of 1/OS_CFG_TMR_TASK_RATE_HZ of a second (see os_cfg_app.h).

(8) The .Dly field contains the one-shot time when the timer is configured (i.e., created) as a one-shot timer and the initial delay when the timer is created as a periodic timer. The value is expressed in multiples of 1/OS_CFG_TMR_TASK_RATE_HZ of a second (see os_cfg_app.h).

(9) The .Period field is the timer period when the timer is created to operate in periodic mode. The value is expressed in multiples of 1/OS_CFG_TMR_TASK_RATE_HZ of a second (see os_cfg_app.h).

Copyright 2015 Micrium Inc.

(10) The `.Opt` field contains options that are passed to `OSTmrCreate()`.

(11) The `.State` field represents the current state of the timer (see the figure in Timer States, above).

Even if the internals of the `OS_TMR` data type are understood, the application code should never access any of the fields in this data structure directly. Instead, you should always use the Application Programming Interfaces (APIs) provided with µC/OS-III.

Timer Task

`OS_TmrTask()` is a task created by µC/OS-III (assumes setting `OS_CFG_TMR_EN` to `DEF_ENABLED` in `os_cfg.h`) and its priority is configurable by the user through µC/OS-III's configuration file `os_cfg_app.h` (see `OS_CFG_TMR_TASK_PRIO`). `OS_TmrTask()` is typically set to a medium priority.

`OS_TmrTask()` is a periodic task and uses the same interrupt source used to generate clock ticks. However, timers are generally updated at a slower rate (i.e., typically 10 Hz or so) and thus, the timer tick rate is divided down in software. If the tick rate is 1000 Hz and the desired timer rate is 10 Hz then the timer task will be signaled every 100th tick interrupt as shown in the figure below.

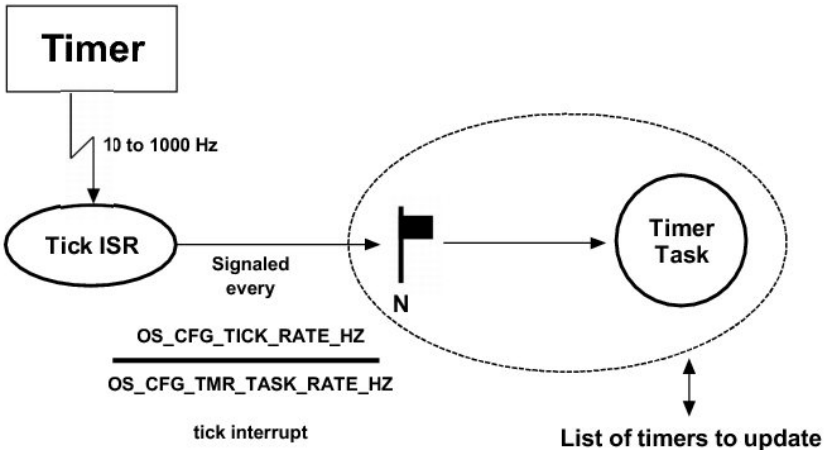


Figure - Tick ISR and Timer Task relationship

The figure below shows timing diagram associated with the timer management task.

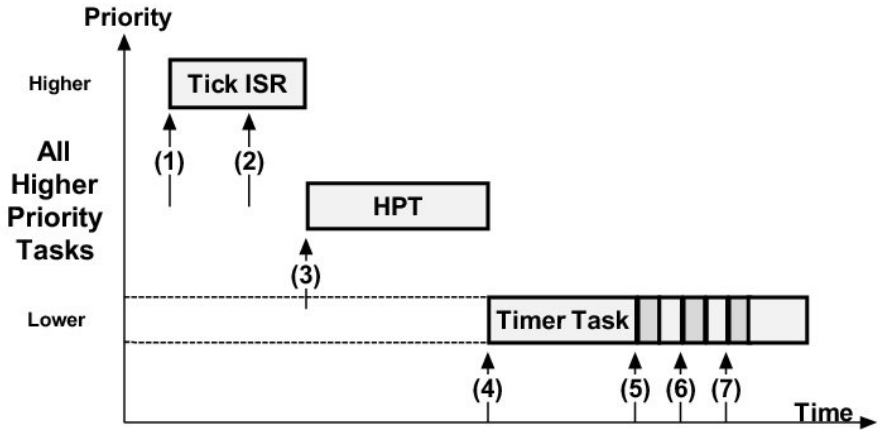


Figure - Timing Diagram

(1) The tick ISR occurs and assumes interrupts are enabled and executes.

(2) The tick ISR signals the tick task that it is time for it to update timers.

(3) The tick ISR terminates, however there might be higher priority tasks that need to execute (assuming the timer task has a lower priority). Therefore, µC/OS-III runs the higher priority task(s).

(4) When all higher priority tasks have executed, µC/OS-III switches to the timer task and determines that there are three timers that expired.

(5) The callback for the first timer is executed.

(6) The callback for the second expired timer is executed.

(7) The callback for the third expired timer is executed.

There are a few interesting things to notice:

- Execution of the callback functions is performed within the context of the timer task. This means that the application code will need to make sure there is sufficient stack space for the timer task to handle these callbacks.

Copyright 2015 Micrium Inc.

- The callback functions are executed one after the other based on the order they are found in the timer list.

- The execution time of the timer task greatly depends on how many timers expire and how long each of the callback functions takes to execute. Since the callbacks are provided by the application code they have a large influence on the execution time of the timer task.

- The timer callback functions must never wait on events because this would delay the timer task for excessive amounts of time, if not forever.

- Callbacks should execute as quickly as possible.

Timer List

µC/OS-III applications may require many timers. The timer manager implements a simple linear list where each timer is linked in a doubly linked list as shown in the figure below.

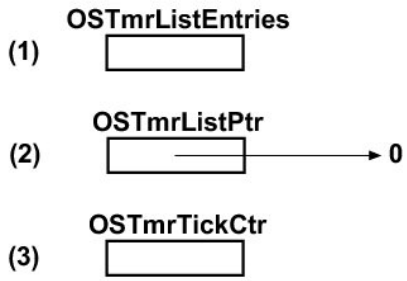


Figure - Empty Timer List

(1) `OSTmrListEntries` contains the current number of entries in the list. This variable is updated whenever timers are added or removed from the list.

(2) `OSTmrListPtr` contains a pointer to a doubly linked list of timers that the timer manager will need to update.

(3) `OSTmrTickCtr` is incremented by `OS_TmrTask()` every time the tick ISR signals the task. This counter basically keeps track of the number of times the timer task has been signaled.

Timers are inserted in the timer list by calling `OSTmrStart()` and, a timer must be created before it can be used. Newly created timers are always inserted at the beginning of the list as shown in the figure following the code listing below and the code listing itself.

```
OS_TMR  MyTmr1;
          OS_TMR  MyTmr2;


          void MyTmrCallbackFnct1 (void *p_arg)
          {
              /* Do something when timer #1 expires */
          }


          void MyTmrCallbackFnct2 (void *p_arg)
          {
              /* Do something when timer #2 expires */
          }


          void MyTask (void *p_arg)
          {
              OS_ERR  err;


              while (DEF_ON) {
                  :
                  OSTmrCreate((OS_TMR            *)&MyTmr1,
                              (OS_CHAR           *)“My Timer #1”,
                              (OS_TICK            )1,
                              (OS_TICK            )0,
                              (OS_OPT             )OS_OPT_TMR_ONE_SHOT,
                              (OS_TMR_CALLBACK_PTR)MyTmrCallbackFnct1,
                              (void              *)0,
                              (OS_ERR            *)&err);
                  /* Check ’err” */
                  OSTmrStart ((OS_TMR *)&MyTmr1,
                              (OS_ERR *)&err);
                  /* Check “err” */
                  // Continues in the next code listing!
```

Listing - Creating and Starting a timer

Since this is the first timer inserted in the timer list, the `.NextPtr` and `.PrevPtr` both point to `NULL`.

Copyright 2015 Micrium Inc.

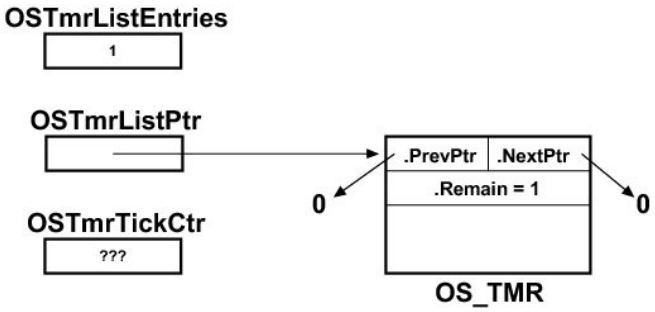


Figure - Inserting a timer in the timer list

The code below shows creating and starting another timer. This is performed “before” the timer task is signaled.

```
// Continuation of code from previous code listing.
                :
                :
                OSTmrCreate((OS_TMR            *)&MyTmr2,
                            (OS_CHAR           *)“My Timer #2”,
                            (OS_TICK            )10,
                            (OS_TICK            )0,
                            (OS_OPT             )OS_OPT_TMR_ONE_SHOT,
                            (OS_TMR_CALLBACK_PTR)MyTmrCallbackFnct2,
                            (void              *)0,
                            (OS_ERR            *)&err);
                /* Check ’err” */
                OSTmrStart ((OS_TMR *)&MyTmr,
                            (OS_ERR *)&err);
                /* Check ’err” */
            }
        }
```

Listing - Creating and Starting a timer - continued

The “second timer” is inserted at the head of the list as shown in the figure below.

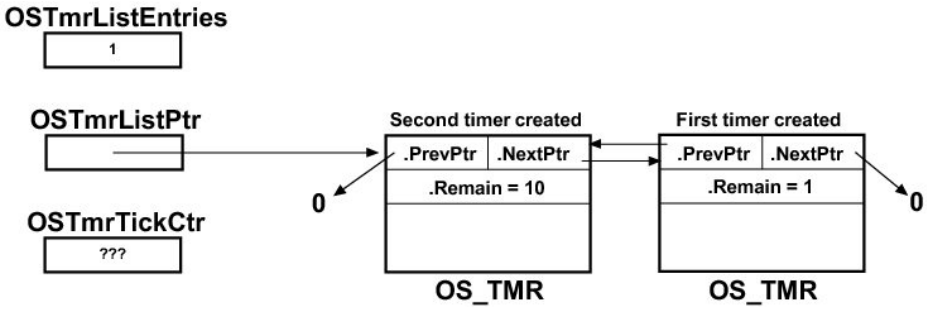


Figure - Inserting a second timer in the tick list

When the timer task executes (see `OS_TmrTask()` in `os_tmr.c`), it starts by incrementing

OSTmrTickCtr and goes through the list (linearly) and decrements each of the .Remain field. When the .Remain field reaches zero, the timer manager executes the callback function associated with the timer and, if the timer is set to periodic, reloads the .Remain field with the time remaining to expiration. If the timer is configured as a one-shot timer then the timer is removed from the list upon expiration.

Timer management occurs at the task level. The list is protected using an internal mutual exclusion semaphore (mutex) when OS_CFG_MUTEX_EN is set to DEF_ENABLED in os_cfg.h or, by locking the scheduler if mutexes are not enabled. It's **highly** recommend that you use (and thus enable) mutexes because locking the scheduler impacts task responsiveness of other, higher priority tasks in your application.

Copyright 2015 Micrium Inc.

Resource Management

This chapter will discuss services provided by µC/OS-III to manage shared resources. A shared resource is typically a variable (static or global), a data structure, table (in RAM), or registers in an I/O device.

When protecting a shared resource it is preferred to use mutual exclusion semaphores, as will be described in this chapter. Other methods are also presented.

Tasks can easily share data when all tasks exist in a single address space and can reference global variables, pointers, buffers, linked lists, ring buffers, etc. Although sharing data simplifies the exchange of information between tasks, it is important to ensure that each task has exclusive access to the data to avoid contention and data corruption.

For example, when implementing a module that performs a simple time-of-day algorithm in software, the module obviously keeps track of hours, minutes and seconds. The `TimeOfDay()` task may appear as shown in the listing below.

Imagine if this task was preempted by another task, after setting the Minutes to 0, because an interrupt occurred and that the other task was more important than the `TimeOfDay()` task. Now imagine what will happen if this higher priority task wants to know the current time from the time-of-day module. Since the Hours were not incremented prior to the interrupt, the higher-priority task will read the time incorrectly and, in this case, it will be incorrect by a whole hour.

The code that updates variables for the `TimeOfDay()` task must treat all of the variables indivisibly (or atomically) whenever there is possible preemption. Time-of-day variables are considered shared resources and any code that accesses those variables must have exclusive access through what is called a critical section. µC/OS-III provides services to protect shared resources and enables the easy creation of critical sections.

Copyright 2015 Micrium Inc.

```
CPU_INT08U  Hours;
          CPU_INT08U  Minutes;
          CPU_INT08U  Seconds;


          void  TimeOfDay (void *p_arg)
          {
              OS_ERR  err;


              (void)&p_arg;
              while (DEF_ON) {
                  OSTimeDlyHMSM(0,
                                0,
                                1,
                                0,
                                OS_OPT_TIME_HMSM_STRICT,
                                &err);
                  /* Examine “err” to make sure the call was successful */
                  Seconds++;
                  if (Seconds > 59) {
                      Seconds = 0;
                      Minutes++;
                      if (Minutes > 59) {
                          Minutes = 0;
                          Hours++;
                          if (Hours > 23) {
                              Hours = 0;
                          }
                      }
                  }
              }
          }
```

Listing - Faulty Time-Of-Day clock task

The most common methods of obtaining exclusive access to shared resources and to create *critical sections* are:

- disabling interrupts
- disabling the scheduler
- using semaphores
- using mutual exclusion semaphores (a.k.a. a mutex)

The mutual exclusion mechanism used depends on how fast the code will access a shared resource, as shown in the table below.

Copyright 2015 Micrium Inc.

Resource Sharing Method	When should you use?
Disable/Enable Interrupts	When access to shared resource is very quick (reading from or writing to few variables) and access is faster than µC/OS-III's interrupt disable time. It is highly recommended to not use this method as it impacts interrupt latency.
Locking/Unlocking the Scheduler	When access time to the shared resource is longer than µC/OS-III's interrupt disable time, but shorter than µC/OS-III's scheduler lock time. Locking the scheduler has the same effect as making the task that locks the scheduler the highest-priority task. It is recommended not to use this method since it defeats the purpose of using µC/OS-III. However, it is a better method than disabling interrupts, as it does not impact interrupt latency.
Semaphores	When all tasks that need to access a shared resource do not have deadlines. This is because semaphores may cause unbounded priority inversions (described later). However, semaphore services are slightly faster (in execution time) than mutual-exclusion semaphores.
Mutual Exclusion Semaphores	This is the preferred method for accessing shared resources, especially if the tasks that need to access a shared resource have deadlines. µC/OS-III's mutual exclusion semaphores have a built-in priority inheritance mechanism, which avoids unbounded priority inversions. However, mutual exclusion semaphore services are slightly slower (in execution time) than semaphores since the priority of the owner may need to be changed, which requires CPU processing.

Table - Resource sharing

Disable-Enable Interrupts

The easiest and fastest way to gain exclusive access to a shared resource is by disabling and enabling interrupts, as shown in the pseudo-code in the listing below.

```
Disable Interrupts;
             Access the resource;
             Enable  Interrupts;
```

Listing - Disabling and Enabling Interrupts

µC/OS-III uses this technique (as do most, if not all, kernels) to access certain internal variables and data structures, ensuring that these variables and data structures are manipulated atomically. However, disabling and enabling interrupts are actually CPU-related functions rather than OS-related functions and functions in CPU-specific files are provided to accomplish this (see the `cpu.h` file of the processor being used). The services provided in the CPU module are called µC/CPU. Each different target CPU architecture has its own set of µC/CPU-related files.

```
void YourFunction (void)
          {
              CPU_SR_ALLOC();                (1)


              CPU_CRITICAL_ENTER();          (2)
              Access the resource;           (3)
              CPU_CRITICAL_EXIT();           (4)
          }
```

Listing - Using CPU macros to disable and enable interrupts

(1) The `CPU_SR_ALLOC()` macro is required when the other two macros that disable/enable interrupts are used. This macro simply allocates storage for a local variable to hold the value of the current interrupt disable status of the CPU. If interrupts are already disabled we do not want to enable them upon exiting the critical section.

(2) `CPU_CRITICAL_ENTER()` saves the current state of the CPU interrupt disable flag(s) in the local variable allocated by `CPU_SR_ALLOC()` and disables all maskable interrupts.

(3) The critical section of code is then accessed without fear of being changed by either an

Copyright 2015 Micrium Inc.

ISR or another task because interrupts are disabled. In other words, this operation is now atomic.

(4) `CPU_CRITICAL_EXIT()` restores the previously saved interrupt disable status of the CPU from the local variable.

`CPU_CRITICAL_ENTER()` and `CPU_CRITICAL_EXIT()` are always used in pairs. Interrupts should be disabled for as short a time as possible as disabling interrupts impacts the response of the system to interrupts. This is known as interrupt latency. Disabling and enabling is used only when changing or copying a few variables.

Note that this is the only way that a task can share variables or data structures with an ISR.

µC/CPU provides a way to actually measure interrupt latency.

When using µC/OS-III, interrupts may be disabled for as much time as µC/OS-III does, without affecting *interrupt latency*. Obviously, it is important to know how long µC/OS-III disables interrupts, which depends on the CPU used.

Although this method works, you should avoid disabling interrupts as it affects the responsiveness of the system to real-time events.

Copyright 2015 Micrium Inc.

Lock-Unlock

If the task does not share variables or data structures with an ISR, you can disable and enable µC/OS-III's scheduler while accessing the resource, as shown in the listing below.

```
void YourFunction (void)
          {
              OS_ERR  err;                    (1)


              OSSchedLock(&err);              (2)
              Access the resource;            (3)
              OSSchedUnlock(&err);            (4)
          }
```

Listing - Accessing a resource with the scheduler locked

Using this method, two or more tasks share data without the possibility of contention. Note that while the scheduler is locked, interrupts are enabled and if an interrupt occurs while in the critical section, the ISR is executed immediately. At the end of the ISR, the kernel always returns to the interrupted task even if a higher priority task is made ready-to-run by the ISR. Since the ISR returns to the interrupted task, the behavior of the kernel is similar to that of a non-preemptive kernel (while the scheduler is locked).

`OSSchedLock()` and `OSSchedUnlock()` can be nested up to 250 levels deep. The scheduler is invoked only when `OSSchedUnlock()` is called the same number of times the application called `OSSchedLock()`.

After the scheduler is unlocked, µC/OS-III performs a context switch if a higher priority task is ready-to-run.

µC/OS-III will not allow the user to make blocking calls when the scheduler is locked. If the application were able to make blocking calls, the application would most likely fail.

Although this method works well, you can avoid disabling the scheduler as it defeats the purpose of having a preemptive kernel. Locking the scheduler makes the current task the highest priority task.

Copyright 2015 Micrium Inc.

Semaphores in uC-OS-III

A semaphore was originally a mechanical signaling mechanism. The railroad industry used the device to provide a form of mutual exclusion for railroads tracks shared by more than one train. In this form, the semaphore signaled trains by closing a set of mechanical arms to block a train from a section of track that was currently in use. When the track became available, the arm would swing up and the waiting train would then proceed.

The notion of using a semaphore in software as a means of mutual exclusion was invented by the Dutch computer scientist Edgser Dijkstra in 1959. In computer software, a semaphore is a protocol mechanism offered by most multitasking kernels. Semaphores, originally used to control access to shared resources, but now they are used for synchronization as described in Synchronization. However, it is useful to describe how semaphores can be used to share resources. The pitfalls of semaphores will be discussed in a later section.

A semaphore was originally a “lock mechanism” and code acquired the key to this lock to continue execution. Acquiring the key means that the executing task has permission to enter the section of otherwise locked code. Entering a section of locked code causes the task to wait until the key becomes available.

Typically, two types of semaphores exist: binary semaphores and counting semaphores. As its name implies, a binary semaphore can only take two values: 0 or 1. A counting semaphore allows for values between 0 and 255, 65,535, or 4,294,967,295, depending on whether the semaphore mechanism is implemented using 8, 16, or 32 bits, respectively. For C/OS-III, the maximum value of a semaphore is determined by the data type `OS_SEM_CTR` (see `os_type.h`), which can be changed as needed. Along with the semaphore’s value, C/OS-III also keeps track of tasks waiting for the semaphore’s availability.

Only tasks are allowed to use semaphores when semaphores are used for sharing resources; ISRs are not allowed.

A semaphore is a kernel object defined by the `OS_SEM` data type, which is defined by the structure `os_sem` (see `os.h`). The application can have any number of semaphores (limited only by the amount of RAM available).

There are a number of operations the application is able to perform on semaphores, summarized in the table below. In this chapter, only the three most used functions are discussed: `OSSemCreate()`, `OSSemPend()`, and `OSSemPost()`. The other functions are described in

Copyright 2015 Micrium Inc.

µC/OS-III API Reference. When semaphores are used for sharing resources, every semaphore function must be called from a task and never from an ISR. The same limitation does not apply when using semaphores for signaling.

Function Name	Operation
`OSSemCreate()`	Create a semaphore.
`OSSemDel()`	Delete a semaphore.
`OSSemPend()`	Wait on a semaphore.
`OSSemPendAbort()`	Abort the wait on a semaphore.
`OSSemPost()`	Release or signal a semaphore.
`OSSemSet()`	Force the semaphore count to a desired value.

Copyright 2015 Micrium Inc.

Binary Semaphores

A task that wants to acquire a resource must perform a Wait (or Pend) operation. If the semaphore is available (the semaphore value is greater than 0), the semaphore value is set to `0`, and the task continues execution (owning the resource). If the semaphore's value is 0, the task performing a Wait on the semaphore is placed in a waiting list. µC/OS-III allows a timeout to be specified. If the semaphore is not available within a certain amount of time, the requesting task is made ready-to-run, and an error code (indicating that a timeout has occurred) is returned to the caller.

A task releases a semaphore by performing a Signal (or Post) operation. If no task is waiting for the semaphore, the semaphore value is simply set to `1`. If there is at least one task waiting for the semaphore, the highest-priority task waiting on the semaphore is made ready-to-run, and the semaphore value is not incremented. If the readied task has a higher priority than the current task (the task releasing the semaphore), a context switch occurs and the higher-priority task resumes execution. The current task is suspended until it again becomes the highest-priority task that is ready-to-run.

The operations described above are summarized using the pseudo-code shown in the listing below.

```
OS_SEM  MySem;                              (1)


void  main (void)
{
    OS_ERR  err;
    :
    :
    OSInit(&err);
    :
    OSSemCreate(&MySem,                     (2)
                "My Semaphore",             (3)
                1,                          (4)
                &err);                      (5)
    /* Check "err" */
    :
    /* Create task(s) */
    :
    OSStart(&err);
    (void)err;
}
```

Listing - Using a semaphore to access a shared resource

(1) The application must declare a semaphore as a variable of type `OS_SEM`. This variable will be referenced by other semaphore services.

(2) You create a semaphore by calling `OSSemCreate()` and pass the address to the semaphore allocated in (1). The semaphore must be created before it can be used by other tasks. Here, the semaphore is initialized in startup code (i.e., `main()`), however it could also be initialized by a task (but it must be initialized before it is used).

(3) You can assign an ASCII name to the semaphore, which can be used by debuggers or µC/Probe to easily identify the semaphore. Storage for the ASCII characters is typically in ROM, which is typically more plentiful than RAM. If it is necessary to change the name of the semaphore at runtime, you can store the characters in an array in RAM and simply pass the address of the array to `OSSemCreate()`. Of course, the array must be NUL terminated.

(4) You specify the initial value of the semaphore. You should initialize the semaphore to 1 when the semaphore is used to access a single shared resource (as in this example).

(5) `OSSemCreate()` returns an error code based on the outcome of the call. If all the arguments are valid, `err` will contain `OS_ERR_NONE`. Refer to the description of `OSSemCreate()` in Appendix A, µC/OS-III API Reference for a list of other error codes and their meaning.

Copyright 2015 Micrium Inc.

```
void  Task1 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;


    while (DEF_ON) {
        :
        OSSemPend(&MySem,                               (1)
                  0,                                    (2)
                  OS_OPT_PEND_BLOCKING,                 (3)
                  &ts,                                  (4)
                  &err);                                (5)
        switch (err) {
            case OS_ERR_NONE:
                 Access Shared Resource;                (6)
                 OSSemPost(&MySem,                      (7)
                           OS_OPT_POST_1,               (8)
                           &err);                       (9)
                 /* Check “err” */
                 break;

            case OS_ERR_PEND_ABORT:
                 /* The pend was aborted by another task     */
                 break;

            case OS_ERR_OBJ_DEL:
                 /* The semaphore was deleted                */
                 break;

            default:
                 /* Other errors                             */
        }
        :
    }
}
```

Listing - Using a semaphore to access a shared resource

(1) The task pends (or waits) on the semaphore by calling `OSSemPend()`. The application must specify the desired semaphore to wait upon, and the semaphore must have been previously created.

(2) The next argument is a timeout specified in number of clock ticks. The actual timeout depends on the tick rate. If the tick rate (see `os_cfg_app.h`) is set to 1000, a timeout of 10 ticks represents 10 milliseconds. Specifying a timeout of zero (0) means waiting forever for the semaphore.

(3) The third argument specifies how to wait. There are two options: `OS_OPT_PEND_BLOCKING` and `OS_OPT_PEND_NON_BLOCKING`. The blocking option means that if the semaphore is not available, the task calling `OSSemPend()` will wait until the semaphore is posted or until the timeout expires. The non-blocking option indicates that if the semaphore is not

Copyright 2015 Micrium Inc.

available, OSSemPend() will return immediately and not wait. This last option is rarely used when using a semaphore to protect a shared resource.

(4) When the semaphore is posted, µC/OS-III reads a “timestamp” and returns this timestamp when OSSemPend() returns. This feature allows the application to know “when” the post happened and the semaphore was released. At this point, OS_TS_GET() is read to get the current timestamp and you can compute the difference, indicating the length of the wait.

(5) OSSemPend() returns an error code based on the outcome of the call. If the call is successful, err will contain OS_ERR_NONE. If not, the error code will indicate the reason for the error. See Appendix A, µC-OS-III Configuration Manual for a list of possible error code for OSSemPend(). Checking for error return values is important since other tasks might delete or otherwise abort the pend. However, it is not a recommended practice to delete kernel objects at run time as the action may cause serious problems.

(6) The resource can be accessed when OSSemPend() returns, if there are no errors.

(7) When finished accessing the resource, you simply call OSSemPost() and specify the semaphore to be released.

(8) OS_OPT_POST_1 indicates that the semaphore is signaling a single task, if there are many tasks waiting on the semaphore. In fact, you should always specify this option when a semaphore is used to access a shared resource.

(9) As with most µC/OS-III functions, you specify the address of a variable that will receive an error message from the call.

Copyright 2015 Micrium Inc.

```
void  Task2 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;


    while (DEF_ON) {
        :
        OSSemPend(&MySem,                              (1)
                  0,
                  OS_OPT_PEND_BLOCKING,
                  &ts,
                  &err);
        switch (err) {
            case OS_ERR_NONE:
                 Access Shared Resource;
                 OSSemPost(&MySem,
                           OS_OPT_POST_1,
                           &err);
                 /* Check “err” */
                 break;

            case OS_ERR_PEND_ABORT:
                 /* The pend was aborted by another task     */
                 break;

            case OS_ERR_OBJ_DEL:
                 /* The semaphore was deleted                */
                 break;

            default:
                 /* Other errors                             */
        }
        :
    }
}
```

Listing - Using a semaphore to access a shared resource

(1) Another task wanting to access the shared resource needs to use the same procedure to access the shared resource.

Semaphores are especially useful when tasks share I/O devices. Imagine what would happen if two tasks were allowed to send characters to a printer at the same time. The printer would contain interleaved data from each task. For instance, the printout from Task 1 printing “I am Task 1,” and Task 2 printing “I am Task 2,” could result in “I Ia amm T Tasask k1 2”. In this case, you can use a semaphore and initialize it to 1 (i.e., a binary semaphore). The rule is simple: to access the printer each task must first obtain the resource’s semaphore. The figure below shows tasks competing for a semaphore to gain exclusive access to the printer. Note that a key, indicating that each task must obtain this key to use the printer, represents the semaphore symbolically.

Copyright 2015 Micrium Inc.

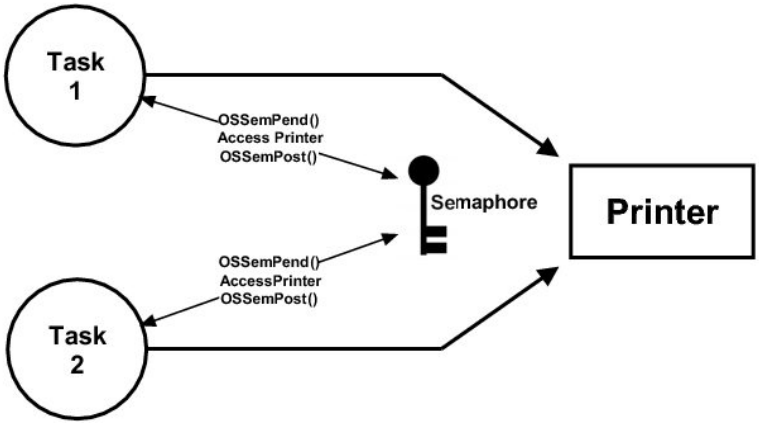


Figure - Using a semaphore to access a printer

The above example implies that each task knows about the existence of the semaphore to access the resource. It is almost always better to encapsulate the critical section and its protection mechanism. Each task would therefore not know that it is acquiring a semaphore when accessing the resource. For example, an RS-232C port is used by multiple tasks to send commands and receive responses from a device connected at the other end as shown in the figure below.

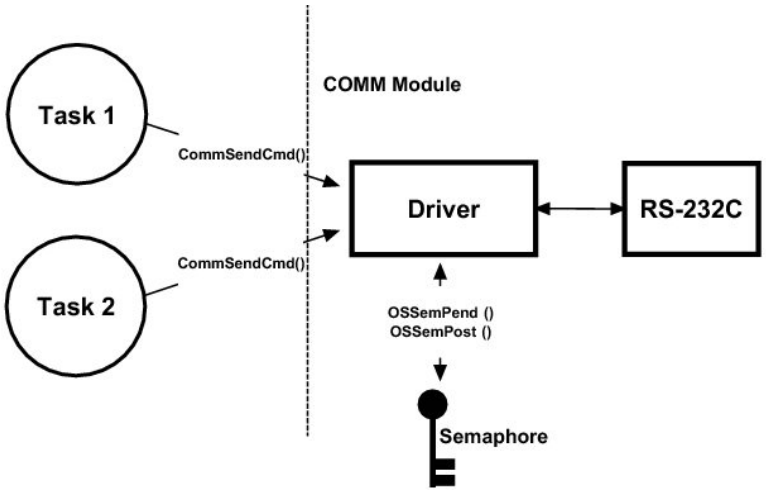


Figure - Hiding a semaphore from a task

The function `CommSendCmd()` is called with three arguments: the ASCII string containing the command, a pointer to the response string from the device, and finally, a timeout in case the device does not respond within a certain amount of time. The pseudo-code for this function is shown in the listing below.

```
APP_ERR  CommSendCmd (CPU_CHAR  *cmd,
                      CPU_CHAR  *response,
                      OS_TICK    timeout)
{
    Acquire serial port’s semaphore;
    Send “cmd” to device;
    Wait for response with “timeout”;
    if (timed out) {
        Release serial port’s semaphore;
        return (error code);
    } else {
        Release serial port’s semaphore;
        return (no error);
    }
}
```

Listing - Encapsulating the use of a semaphore

Each task that needs to send a command to the device must call this function. The semaphore is assumed to be initialized to 1 (i.e., available) by the communication driver initialization routine. The first task that calls `CommSendCmd()` acquires the semaphore, proceeds to send the command, and waits for a response. If another task attempts to send a command while the port is busy, this second task is suspended until the semaphore is released. The second task appears simply to have made a call to a normal function that will not return until the function performs its duty. When the semaphore is released by the first task, the second task acquires the semaphore and is allowed to use the RS-232C port.

Copyright 2015 Micrium Inc.

Counting Semaphores

A counting semaphore is used when elements of a resource can be used by more than one task at the same time. For example, a counting semaphore is used in the management of a buffer pool, as shown in the figure below. Let's assume that the buffer pool initially contains 10 buffers. A task obtains a buffer from the buffer manager by calling `BufReq()`. When the buffer is no longer needed, the task returns the buffer to the buffer manager by calling `BufRel()`. The pseudo-code for these functions is shown in the listing that follows the figure below.

The buffer manager satisfies the first 10 buffer requests because the semaphore is initialized to 10. When all buffers are used, a task requesting a buffer is suspended until a buffer becomes available. You use µC/OS-III's `OSMemGet()` and `OSMemPut()` (see Memory Management API Changes) to obtain a buffer from the buffer pool. When a task is finished with the buffer it acquired, the task calls `BufRel()` to return the buffer to the buffer manager and the buffer is inserted into the linked list before the semaphore is signaled. By encapsulating the interface to the buffer manager in `BufReq()` and `BufRel()`, the caller does not need to be concerned with actual implementation details.

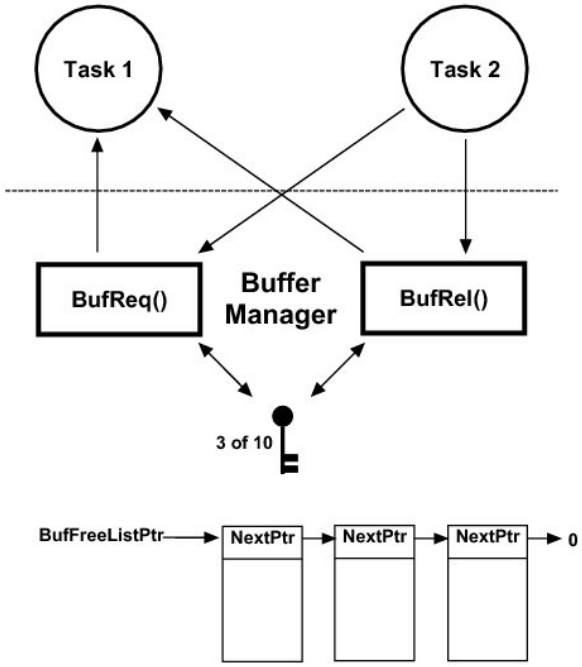


Figure - Using a counting semaphore

```
BUF  *BufReq (void)
          {
              BUF  *ptr;


              Wait on semaphore;
              ptr = OSMemGet(...) ;                    /* Get a buffer      */
              return (ptr);
          }


          void  BufRel (BUF *ptr)
          {
              OSMemPut(..., (void *)ptr, ...);         /* Return the buffer */
              Signal semaphore;
          }
```

Listing - Buffer management using a semaphore.

Note that the details of creating the memory partition are removed since this is discussed in Memory Management API Changes. The semaphore is used here to extend the memory management capabilities of µC/OS-III, and to provide it with a blocking mechanism. However, only tasks can make `BufReq()` and `BufRel()` calls.

Notes on Semaphores

Using a semaphore to access a shared resource does not increase interrupt latency. If an ISR or the current task makes a higher priority task ready-to-run while accessing shared data, the higher priority task executes immediately.

An application may have as many semaphores as required to protect a variety of different resources. For example, one semaphore may be used to access a shared display, another to access a shared printer, another for shared data structures, and another to protect a pool of buffers, etc. However, it is preferable to use semaphores to protect access to I/O devices rather than memory locations.

Semaphores are often overused. The use of a semaphore to access a simple shared variable is overkill in most situations. The overhead involved in acquiring and releasing the semaphore consumes valuable CPU time. You can perform the job more efficiently by disabling and enabling interrupts, however there is an indirect cost to disabling interrupts: even higher priority tasks that do not share the specific resource are blocked from using the CPU. Suppose, for instance, that two tasks share a 32-bit integer variable. The first task increments the variable, while the second task clears it. When considering how long a processor takes to perform either operation, it is easy to see that a semaphore is not required to gain exclusive access to the variable. Each task simply needs to disable interrupts before performing its operation on the variable and enable interrupts when the operation is complete. A semaphore should be used if the variable is a floating-point variable and the microprocessor does not support hardware floating-point operations. In this case, the time involved in processing the floating-point variable may affect interrupt latency if interrupts are disabled.

Semaphores are subject to a serious problem in real-time systems called priority inversion, which is described in Priority Inversions.

Copyright 2015 Micrium Inc.

Semaphore Internals

As previously mentioned, a semaphore is a kernel object as defined by the OS_SEM data type, which is derived from the structure os_sem (see os.h) as shown in the listing below.

The services provided by µC/OS-III to manage semaphores are implemented in the file os_sem.c. Semaphore services are enabled at compile time by setting the configuration constant OS_CFG_SEM_EN to 1 in os_cfg.h.

```
typedef  struct  os_sem  OS_SEM;             (1)


          struct  os_sem {
              OS_OBJ_TYPE          Type;               (2)
              CPU_CHAR            *NamePtr;            (3)
              OS_PEND_LIST         PendList;           (4)
              OS_SEM_CTR           Ctr;                (5)
              CPU_TS               TS;                 (6)
          };
```

Listing - OS_SEM data type

(1) In µC/OS-III, all structures are given a data type. All data types start with "OS_" and are uppercase. When a semaphore is declared, you simply use OS_SEM as the data type of the variable used to declare the semaphore.

(2) The structure starts with a "Type" field, which allows it to be recognized by µC/OS-III as a semaphore. Other kernel objects will also have a ".Type" as the first member of the structure. If a function is passed a kernel object, µC/OS-III will confirm that it is being passed the proper data type (assuming OS_CFG_OBJ_TYPE_CHK_EN is set to 1 in os_cfg.h). For example, if you pass a message queue (OS_Q) to a semaphore service (for example OSSemPend()), µC/OS-III will recognize that an invalid object was passed, and return an error code accordingly.

(3) Each kernel object can be given a name for easier recognition by debuggers or µC/Probe. This member is simply a pointer to an ASCII string, which is assumed to be NUL terminated.

(4) Since it is possible for multiple tasks to wait (or pend) on a semaphore, the semaphore object contains a pend list as described in Pend Lists.

Copyright 2015 Micrium Inc.

(5) A semaphore contains a counter. As explained above, the counter can be implemented as either an 8-, 16- or 32-bit value, depending on how the data type `OS_SEM_CTR` is declared in `os_type.h`.

µC/OS-III does not make a distinction between binary and counting semaphores. The distinction is made when the semaphore is created. If creating a semaphore with an initial value of 1, it is a binary semaphore. When creating a semaphore with a value > 1, it is a counting semaphore. In the next chapter, you will discover that a semaphore is more often used as a signaling mechanism and therefore, the semaphore counter is initialized to zero.

(6) A semaphore contains a timestamp used to indicate the last time the semaphore was posted. µC/OS-III assumes the presence of a free-running counter that allows the application to make time measurements. When the semaphore is posted, the free-running counter is read and the value is placed in this field, which is returned when `OSSemPend()` is called. The value of this field is more useful when a semaphore is used as a signaling mechanism (see Synchronization), as opposed to a resource-sharing mechanism.

Even if the user understands the internals of the `OS_SEM` data type, the application code should never access any of the fields in this data structure directly. Instead, you should always use the APIs provided with µC/OS-III.

As previously mentioned, semaphores must be created before they can be used by an application.

A task waits on a semaphore before accessing a shared resource by calling `OSSemPend()` as shown in the listing below.

Copyright 2015 Micrium Inc.

```
OS_SEM  MySem;


          void MyTask (void *p_arg)
          {
              OS_ERR  err;
              CPU_TS  ts;

              :
              while (DEF_ON) {
                  :
                  OSSemPend(&MySem,               /* (1) Pointer to semaphore                          */
                            10,                   /*     Wait up until this time for the semaphore     */
                            OS_OPT_PEND_BLOCKING, /*     Option(s)                                     */
                            &ts,                  /*     Returned timestamp of when sem. was released  */
                            &err);                /*     Pointer to Error returned                     */
                  :
                  /* Check “err” */               /* (2)                                               */
                  :
                  OSSemPost(&MySem,               /* (3) Pointer to semaphore                          */
                            OS_OPT_POST_1,        /*     Option(s) … always OS_OPT_POST_1              */
                            &err);                /*     Pointer to Error returned                     */
                  /* Check “err” */
                  :
                  :
              }
          }
```

Listing - Pending on and Posting to a Semaphore

(1) When called, `OSSemPend()` starts by checking the arguments passed to this function to make sure they have valid values (assuming `OS_CFG_ARG_CHK_EN` is set to `1` in `os_cfg.h`).

If the semaphore counter (`.Ctr` of `OS_SEM`) is greater than zero, the counter is decremented and `OSSemPend()` returns. If `OSSemPend()` returns without error, then the task now owns the shared resource.

If the semaphore counter is zero, then another task owns the semaphore, and the calling task will need to wait for the semaphore to be released. If you specify `OS_OPT_PEND_NON_BLOCKING` as the option (the application does not want the task to block), `OSSemPend()` returns immediately to the caller and the returned error code indicates that the semaphore is unavailable. You use this option if the task does not want to wait for the resource to be available, and would prefer to do something else and check back later.

If you specify the `OS_OPT_PEND_BLOCKING` option, the calling task will be inserted in the list of tasks waiting for the semaphore to become available. The task is inserted in the list by priority order and therefore, the highest priority task waiting on the semaphore is

Copyright 2015 Micrium Inc.

at the beginning of the list.

If you specify a non-zero timeout, the task will also be inserted in the tick list. A zero value for a timeout indicates that the user is willing to wait forever for the semaphore to be released. Most of the time, you would specify an infinite timeout when using the semaphore in resource sharing. Adding a timeout may temporarily break a deadlock, however, there are better ways of preventing deadlock at the application level (e.g., never hold more than one semaphore at the same time; resource ordering; etc.).

Assuming blocking, the scheduler is called since the current task is no longer able to run (it is waiting for the semaphore to be released). The scheduler will then run the next highest-priority task that is ready-to-run.

When the semaphore is released and the task that called `OSSemPend()` is again the highest-priority task, µC/OS-III examines the task status to determine the reason why `OSSemPend()` is returning to its caller. The possibilities are:

1) The semaphore was given to the waiting task. This is the preferred outcome.

2) The pend was aborted by another task

3) The semaphore was not posted within the specified timeout

4) The semaphore was deleted

When `OSSemPend()` returns, the caller is notified of the above outcome through an appropriate error code.

(2) If `OSSemPend()` returns with `err` set to `OS_ERR_NONE`, your code can assume that it now has access to the resource.

If `err` contains anything else, `OSSemPend()` either timed out (if the timeout argument was non-zero), the pend was aborted by another task, or the semaphore was deleted by another task. It is always important to examine the returned error code and not assume that everything went well.

(3) When the task is finished accessing the resource, it needs to call `OSSemPost()` and specify the same semaphore. Again, `OSSemPost()` starts by checking the arguments passed to this

Copyright 2015 Micrium Inc.

function to make sure there are valid values (assuming OS_CFG_ARG_CHK_EN is set to 1 in os_cfg.h).

OSSemPost() then calls OS_TS_GET() to obtain the current timestamp so it can place that information in the semaphore to be used by OSSemPend(). This feature is not as useful when semaphores are used to share resources as it is when used as a signaling mechanism.

OSSemPost() checks to see if any tasks are waiting for the semaphore. If not, OSSemPost() simply increments p_sem->Ctr, saves the timestamp in the semaphore, and returns.

If there are tasks waiting for the semaphore to be released, OSSemPost() extracts the highest-priority task waiting for the semaphore. This is a fast operation as the pend list is sorted by priority order.

When calling OSSemPost(), it is possible to specify as an option to not call the scheduler. This means that the post is performed, but the scheduler is not called even if a higher priority task waits for the semaphore to be released. This allows the calling task to perform other post functions (if needed) and make all posts take effect simultaneously without the possibility of context switching in between each post.

Copyright 2015 Micrium Inc.

Priority Inversions

Priority inversion is a problem in real-time systems, and occurs only when using a priority-based preemptive kernel. The figure below illustrates a priority-inversion scenario. Task H (high priority) has a higher priority than Task M (medium priority), which in turn has a higher priority than Task L (low priority).

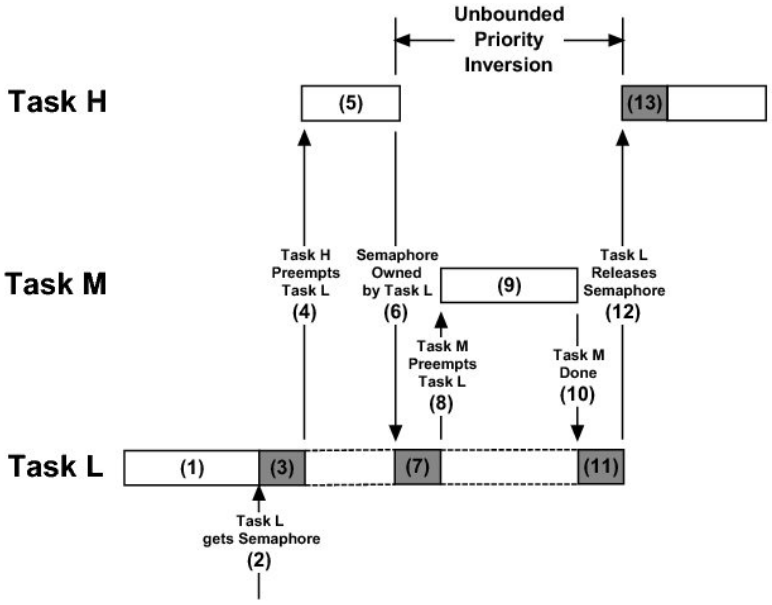


Figure - Unbounded priority Inversion

(1) Task H and Task M are both waiting for an event to occur and Task L is executing.

(2) At some point, Task L acquires a semaphore, which it needs before it can access a shared resource.

(3) Task L performs operations on the acquired resource.

(4) The event that Task H was waiting for occurs, and the kernel suspends Task L and start executing Task H since Task H has a higher priority.

(5) Task H performs computations based on the event it just received.

(6) Task H now wants to access the resource that Task L currently owns (i.e., it attempts to

Copyright 2015 Micrium Inc.

get the semaphore that Task L owns). Because Task L owns the resource, Task H is placed in a list of tasks waiting for the semaphore to be available.

(7) Task L is resumed and continues to access the shared resource.

(8) Task L is preempted by Task M since the event that Task M was waiting for occurred.

(9) Task M handles the event.

(10) When Task M completes, the kernel relinquishes the CPU back to Task L.

(11) Task L continues accessing the resource.

(12) Task L finally finishes working with the resource and releases the semaphore. At this point, the kernel knows that a higher-priority task is waiting for the semaphore, and a context switch takes place to resume Task H.

(13) Task H has the semaphore and can access the shared resource.

So, what happened here is that the priority of Task H has been reduced to that of Task L since it waited for the resource that Task L owned. The trouble begins when Task M preempted Task L, further delaying the execution of Task H. This is called an *unbounded priority inversion*. It is unbounded because any medium priority can extend the time Task H has to wait for the resource. Technically, if all medium-priority tasks have known worst-case periodic behavior and bounded execution times, the priority inversion time is computable. This process, however, may be tedious and would need to be revised every time the medium priority tasks change.

This situation can be corrected by raising the priority of Task L, only during the time it takes to access the resource, and restore the original priority level when the task is finished. The priority of Task L should be raised up to the priority of Task H. In fact, µC/OS-III contains a special type of semaphore that does just that and is called a mutual-exclusion semaphore.

Copyright 2015 Micrium Inc.

Mutual Exclusion Semaphores

µC/OS-III supports a special type of binary semaphore called a mutual exclusion semaphore (also known as a mutex) that eliminates unbounded priority inversions. The figure below shows how priority inversions are bounded using a Mutex.

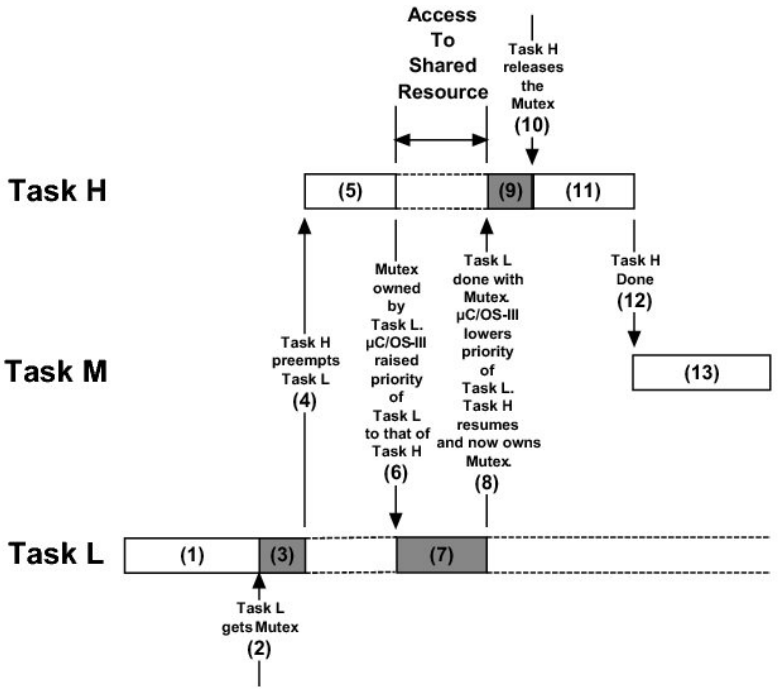

Access To Shared Resource
Task H releases the Mutex (10)
Task H
(5)
(9)
(11)
Task M
Task H preempts Task L (4)
Mutex owned by Task L. µC/OS-III raised priority of Task L to that of Task H (6)
Task L done with Mutex. µC/OS-III lowers priority of Task L. Task H resumes and now owns Mutex. (8)
Task H Done (12)
(13)
Task L
(1)
(3)
(7)
Task L gets Mutex (2)

Figure - Using a mutex to share a resource

(1) Task H and Task M are both waiting for an event to occur and Task L is executing.

(2) At some point, Task L acquires a mutex, which it needs before it is able to access a shared resource.

(3) Task L performs operations on the acquired resource.

(4) The event that Task H waited for occurs and the kernel suspends Task L and begins executing Task H since Task H has a higher priority.

(5) Task H performs computations based on the event it just received.

Copyright 2015 Micrium Inc.

(6) Task H now wants to access the resource that Task L currently owns (i.e., it attempts to get the mutex from Task L). Given that Task L owns the resource, µC/OS-III raises the priority of Task L to the same priority as Task H to allow Task L to finish with the resource and prevent Task L from being preempted by medium-priority tasks.

(7) Task L continues accessing the resource, however it now does so while it is running at the same priority as Task H. Note that Task H is not actually running since it is waiting for Task L to release the mutex. In other words, Task H is in the mutex wait list.

(8) Task L finishes working with the resource and releases the mutex. µC/OS-III notices that Task L was raised in priority and thus lowers Task L to its original priority. After doing so, µC/OS-III gives the mutex to Task H, which was waiting for the mutex to be released.

(9) Task H now has the mutex and can access the shared resource.

(10) Task H is finished accessing the shared resource, and frees up the mutex.

(11) There are no higher-priority tasks to execute, therefore Task H continues execution.

(12) Task H completes and decides to wait for an event to occur. At this point, µC/OS-III resumes Task M, which was made ready-to-run while Task H or Task L were executing. Task M was made ready-to-run because an interrupt (not shown in figure 13-5) occurred which Task M was waiting for.

(13) Task M executes.

Note that there is no priority inversion, only resource sharing. Of course, the faster Task L accesses the shared resource and frees up the mutex, the better.

µC/OS-III implements full-priority inheritance and therefore if a higher priority requests the resource, the priority of the owner task will be raised to the priority of the new requestor.

A mutex is a kernel object defined by the `OS_MUTEX` data type, which is derived from the structure `os_mutex` (see `os.h`). An application may have an unlimited number of mutexes (limited only by the RAM available).

Copyright 2015 Micrium Inc.

Only tasks are allowed to use mutual exclusion semaphores (ISRs are not allowed).

µC/OS-III enables the user to nest ownership of mutexes. If a task owns a mutex, it can own the same mutex up to 250 times. The owner must release the mutex an equivalent number of times. In several cases, an application may not be immediately aware that it called `OSMutexPend()` multiple times, especially if the mutex is acquired again by calling a function as shown in Listing 13-12.

```
OS_MUTEX          MyMutex;
          SOME_STRUCT       MySharedResource;


          void  MyTask (void *p_arg)
          {
              OS_ERR  err;
              CPU_TS  ts;

              :
              while (DEF_ON) {
                  OSMutexPend((OS_MUTEX *)&MyMutex,                    (1)
                              (OS_TICK   )0,
                              (OS_OPT    )OS_OPT_PEND_BLOCKING,
                              (CPU_TS   *)&ts,
                              (OS_ERR   *)&err);
                  /* Check “err”                                  */   (2)
                  /* Acquire shared resource if no error          */
                  MyLibFunction();                                     (3)
                  OSMutexPost((OS_MUTEX *)&MyMutex,                    (7)
                              (OS_OPT    )OS_OPT_POST_NONE,
                              (OS_ERR   *)&err);
                  /* Check “err”                                  */
              }
          }
          void  MyLibFunction (void)
          {
              OS_ERR  err;
              CPU_TS  ts;


              OSMutexPend((OS_MUTEX *)&MyMutex,                        (4)
                          (OS_TICK   )0,
                          (OS_OPT    )OS_OPT_PEND_BLOCKING,
                          (CPU_TS   *)&ts,
                          (OS_ERR   *)&err);
              /* Check “err”                          */
              /* Access shared resource if no error  */                (5)
              OSMutexPost((OS_MUTEX *)&MyMutex,                        (6)
                          (OS_OPT    )OS_OPT_POST_NONE,
                          (OS_ERR   *)&err);
              /* Check “err”                          */
          }
```

Listing - Nesting calls to OSMutexPend()

(1) A task starts by pending on a mutex to access shared resources. `OSMutexPend()` sets a

nesting counter to 1.

(2) You should check the error return value. If no errors exist, `MyTask()` owns `MySharedResource`.

(3) A function is called that will perform additional work.

(4) The designer of `MyLibFunction()` knows that, to access `MySharedResource`, it must acquire the mutex. Since the calling task already owns the mutex, this operation should not be necessary. However, `MyLibFunction()` could have been called by yet another function that might not need access to `MySharedResource`. µC/OS-III allows nested mutex pends, so this is not a problem. The mutex nesting counter is thus incremented to 2.

(5) `MyLibFunction()` can access the shared resource.

(6) The mutex is released and the nesting counter is decremented back to 1. Since this indicates that the mutex is still owned by the same task, nothing further needs to be done, and `OSMutexPost()` simply returns. `MyLibFunction()` returns to its caller.

(7) The mutex is released again and, this time, the nesting counter is decremented back to 0 indicating that other tasks can now acquire the mutex.

You should always check the return value of `OSMutexPend()` (and any kernel call) to ensure that the function returned because you properly obtained the mutex, and not because the return from `OSMutexPend()` was caused by the mutex being deleted, or because another task called `OSMutexPendAbort()` on this mutex.

As a general rule, do not make function calls in critical sections. All mutual exclusion semaphore calls should be in the leaf nodes of the source code (e.g., in the low level drivers that actually touches real hardware or in other reentrant function libraries).

There are a number of operations that can be performed on a mutex, as summarized in the table below. However, in this chapter, we will only discuss the three functions that are most often used: `OSMutexCreate()`, `OSMutexPend()`, and `OSMutexPost()`. Other functions are described in µC-OS-III API Reference.

Copyright 2015 Micrium Inc.

Function Name	Operation
OSMutexCreate()	Create a mutex.
OSMutexDel()	Delete a mutex.
OSMutexPend()	Wait on a mutex.
OSMutexPendAbort()	Abort the wait on a mutex.
OSMutexPost()	Release a mutex.

Table - Mutex API summary

Mutual Exclusion Semaphore Internals

A mutex is a kernel object defined by the OS_MUTEX data type, which is derived from the structure os_mutex (see os.h) as shown in the listing below:

```
typedef  struct  os_mutex  OS_MUTEX;          (1)

struct  os_mutex {
    OS_OBJ_TYPE          Type;                (2)
    CPU_CHAR            *NamePtr;             (3)
    OS_PEND_LIST         PendList;            (4)
    OS_MUTEX            *MutexGrpNextPtr;     (5)
    OS_TCB              *OwnerTCBPtr;         (6)
    OS_NESTING_CTR       OwnerNestingCtr;     (7)
    CPU_TS               TS;                  (8)
};
```

Listing - OS_MUTEX data type

(1) In µC/OS-III, all structures are given a data type. All data types begin with "OS_" and are uppercase. When a mutex is declared, you simply use OS_MUTEX as the data type of the variable used to declare the mutex.

(2) The structure starts with a "Type" field, which allows it to be recognized by µC/OS-III as a mutex. Other kernel objects will also have a ".Type" as the first member of the structure. If a function is passed a kernel object, µC/OS-III will be able to confirm that it is being passed the proper data type (assuming OS_CFG_OBJ_TYPE_CHK_EN is set to DEF_ENABLED in os_cfg.h). For example, if passing a message queue (OS_Q) to a mutex service (for example OSMutexPend()), µC/OS-III will recognize that the application passed an invalid object and return an error code accordingly.

(3) Each kernel object can be given a name to make them easier to recognize by debuggers or µC/Probe. This member is simply a pointer to an ASCII string, which is assumed to be NUL terminated.

(4) Because it is possible for multiple tasks to wait (or pend) on a mutex, the mutex object contains a pend list as described in Pend Lists.

(5) To propagate the correct priority to the owning tasks and because a task can own more than one mutex, µC/OS-III has to track which mutex is owned by which tasks.

Copyright 2015 Micrium Inc.

(6) If the mutex is owned by a task, it will point to the `OS_TCB` of that task.

(7) µC/OS-III allows a task to “acquire” the same mutex multiple times. In order for the mutex to be released, the owner must release the mutex the same number of times that it was acquired. Nesting can be performed up to usually 250-levels deep.

(8) A mutex contains a timestamp, used to indicate the last time it was released. µC/OS-III assumes the presence of a free-running counter that allows applications to make time measurements. When the mutex is released, the free-running counter is read and the value is placed in this field, which is returned when `OSMutexPend()` returns.

Application code should never access any of the fields in this data structure directly. Instead, you should always use the API provided with µC/OS-III.

A mutual exclusion semaphore (mutex) must be created before it can be used by an application. The listing below shows how to create a mutex.

```
OS_MUTEX  MyMutex;                          (1)


void  MyTask (void *p_arg)
{
    OS_ERR  err;
    :
    :
    OSMutexCreate(&MyMutex,                 (2)
                  “My Mutex”,               (3)
                  &err);                    (4)
    /* Check “err” */
    :
    :
}
```

Listing - Creating a mutex

(1) The application must declare a variable of type `OS_MUTEX`. This variable will be referenced by other mutex services.

(2) You create a mutex by calling `OSMutexCreate()` and pass the address to the mutex allocated in (1).

(3) You can assign an ASCII name to the mutex, which can be used by debuggers or µC/Probe to easily identify this mutex. There are no practical limits to the length of the name since µC/OS-III stores a pointer to the ASCII string, and not to the actual characters that makes up the string.

(4) `OSMutexCreate()` returns an error code based on the outcome of the call. If all the arguments are valid, `err` will contain `OS_ERR_NONE`.

Note that since a mutex is always a binary semaphore, there is no need to initialize a mutex counter.

A task waits on a mutual exclusion semaphore before accessing a shared resource by calling `OSMutexPend()` as shown in the listing below (see µC-OS-III API Reference for details regarding the arguments).

```
OS_MUTEX  MyMutex;


          void MyTask (void *p_arg)
          {
              OS_ERR  err;
              CPU_TS  ts;
              :
              while (DEF_ON) {
                  :
                  OSMutexPend(&MyMutex,             /* (1) Pointer to mutex                          */
                              10,                   /*     Wait up until this time for the mutex     */
                              OS_OPT_PEND_BLOCKING, /*     Option(s)                                 */
                              &ts,                  /*     Timestamp of when mutex was released      */
                              &err);                /*     Pointer to Error returned                 */
                  :
                  /* Check “err”                       (2)                                           */
                  :
                  OSMutexPost(&MyMutex,             /* (3) Pointer to mutex                          */
                              OS_OPT_POST_NONE,
                              &err);                /*     Pointer to Error returned                 */
                  /* Check “err”                                                                     */
                  :
                  :
              }
          }
```

Listing - Pending (or waiting) on a Mutual Exclusion Semaphore

(1) When called, `OSMutexPend()` starts by checking the arguments passed to this function to

make sure they have valid values. This assumes that OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h.

If the mutex is available, OSMutexPend() assumes the calling task is now the owner of the mutex and stores a pointer to the task's OS_TCB in p_mutex->OwnerTCPPtr. It also adds the mutex to the calling task list of owned mutex and sets the mutex nesting counter to 1. OSMutexPend() then returns to its caller with an error code of OS_ERR_NONE.

If the task that calls OSMutexPend() already owns the mutex, OSMutexPend() simply increments a nesting counter. In this case, the error returned will indicate OS_ERR_MUTEX_OWNER . Applications can nest calls to OSMutexPend() up to usually 250-levels deep.

If the mutex is already owned by another task and OS_OPT_PEND_NON_BLOCKING is specified, OSMutexPend() returns since the task is not willing to wait for the mutex to be released by its owner.

If the mutex is owned by a task that currently has a lower priority, µC/OS-III will raise the priority of the owner to match the priority of the current task.

If you specify OS_OPT_PEND_BLOCKING as the option, the calling task will be inserted in the list of tasks waiting for the mutex to be available. The task is inserted in the list by priority order and thus, the highest priority task waiting on the mutex is at the beginning of the list.

If you further specify a non-zero timeout, the task will also be inserted in the tick list. A zero value for a timeout indicates a willingness to wait forever for the mutex to be released.

The scheduler is then called since the current task is no longer able to run (it is waiting for the mutex to be released). The scheduler will then run the next highest-priority task that is ready-to-run.

When the mutex is finally released and the task that called OSMutexPend() is again the highest-priority task, a task status is examined to determine the reason why OSMutexPend() is returning to its caller. The possibilities are:

1) The mutex was given to the waiting task. This is the desired outcome.

Copyright 2015 Micrium Inc.

2) The pend was aborted by another task.

3) The mutex was not posted within the specified timeout.

4) The mutex was deleted.

When `OSMutexPend()` returns, the caller is notified of the outcome through an appropriate error code.

(2) If `OSMutexPend()` returns with `err` set to `OS_ERR_NONE`, assume that the calling task now owns the resource and can proceed with accessing it. If `err` contains anything else, then `OSMutexPend()` either timed out (if the timeout argument was non-zero), was aborted by another task, or the mutex was deleted by another task. It is always important to examine returned error codes and not assume everything went as planned.

If "`err`" is `OS_ERR_MUTEX_NESTING`, then the caller attempted to pend on the same mutex.

(3) When your task is finished accessing the resource, it must call `OSMutexPost()` and specify the same mutex. Again, `OSMutexPost()` starts by checking the arguments passed to this function to make sure they contain valid values (Assuming `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`).

`OSMutexPost()` now calls `OS_TS_GET()` to obtain the current timestamp and place that information in the mutex, which will be used by `OSMutexPend()`.

`OSMutexPost()` decrements the nesting counter and, if still non-zero, `OSMutexPost()` returns to the caller. In this case, the current owner has not fully released the mutex. The error code will be `OS_ERR_MUTEX_NESTING`.

`OSMutexPost()` removes the mutex from the calling tasks mutex group.

If there are no tasks waiting for the mutex, `OSMutexPost()` sets `p_mutex->OwnerTCBPtr` to a `NULL` pointer and clears the mutex nesting counter.

If µC/OS-III had to raise the priority of the mutex owner, µC/OS-III sets the priority of the calling task to the highest priority in the calling task's owned mutex group or its base priority, whichever is higher.

Copyright 2015 Micrium Inc.

The highest-priority task waiting on the mutex is then extracted from the pend list and given the mutex. This is a fast operation since the pend list is sorted by priority.

If the option to `OSMutexPost()` is not `OS_OPT_POST_NO_SCHED` then, the scheduler is called to execute the next highest priority task.

Copyright 2015 Micrium Inc.

Should You Use a Semaphore Instead of a Mutex

A semaphore can be used instead of a mutex if none of the tasks competing for the shared resource have deadlines to be satisfied.

However, if there are deadlines to meet, you should use a mutex prior to accessing shared resources. Semaphores are subject to unbounded priority inversions, while mutex are not.

Copyright 2015 Micrium Inc.

Deadlocks

A *deadlock*, also called a deadly embrace, is a situation in which two tasks are each unknowingly waiting for resources held by the other.

Assume Task T1 has exclusive access to Resource R1 and Task T2 has exclusive access to Resource R2 as shown in the pseudo-code of the listing below.

```
void T1 (void *p_arg)
          {
              while (DEF_ON) {
                  Wait for event to occur;          (1)
                  Acquire M1;                       (2)
                  Access  R1;                       (3)
                  :
                  :
                  \--------  Interrupt!             (4)
                  :
                  :                                 (8)
                  Acquire M2;                       (9)
                  Access  R2;
              }
          }
          void  T2 (void *p_arg)
          {
              while (DEF_ON) {
                  Wait for event to occur;          (5)
                  Acquire M2;                       (6)
                  Access  R2;
                  :
                  :
                  Acquire M1;                       (7)
                  Access  R1;
              }
          }
```

Listing - Deadlock problem

(1) Assume that the event that task T1 is waiting for occurs and T1 is now the highest priority task that must execute.

(2) Task T1 executes and acquires Mutex M1.

(3) Resource R1 is accessed.

(4) An interrupt occurs causing the CPU to switch to task T2 since T2 has a higher priority than task T1.

Copyright 2015 Micrium Inc.

(5) The ISR is the event that task T2 was waiting for and therefore T2 resumes execution.

(6) Task T2 acquires mutex M2 and is able to access resource R2.

(7) Task T2 tries to acquire mutex M1, but µC/OS-III knows that mutex M1 is owned by another task.

(8) µC/OS-III switches back to task T1 because Task T2 can no longer continue. It needs mutex M1 to access resource R1.

(9) Task T1 now tries to access mutex M2 but, unfortunately, mutex M2 is owned by task T2. At this point, the two tasks are deadlocked, neither one can continue because each owns a resource that the other one wants.

Techniques used to avoid deadlocks are for tasks to:

- Acquire all resources before proceeding
- Always acquire resources in the same order
- Use timeouts on pend calls

µC/OS-III allows the calling task to specify a timeout when acquiring a mutex. This feature allows a deadlock to be broken, but the same deadlock may then recur later, or many times later. If the mutex is not available within a certain period of time, the task requesting the resource resumes execution. µC/OS-III returns an error code indicating that a timeout occurred. A return error code prevents the task from thinking it has properly obtained the resource.

The pseudo-code avoids deadlocks by first acquiring all resources as shown in Listing 13-17.

Copyright 2015 Micrium Inc.

```
void T1 (void *p_arg)
          {
              while (DEF_ON) {
                  Wait for event to occur;
                  Acquire M1;
                  Acquire M2;
                  Access  R1;
                  Access  R2;
              }
          }


          void  T2 (void *p_arg)
          {
              while (DEF_ON) {
                  Wait for event to occur;
                  Acquire M1;
                  Acquire M2;
                  Access  R1;
                  Access  R2;
              }
          }
```

Listing - Deadlock avoidance – acquire all first and in the same order

The pseudo-code to acquire all of the mutexes in the same order is shown in the listing below. This is similar to the previous example, except that it is not necessary to acquire all the mutexes first, only to make sure that the mutexes are acquired in the same order for both tasks.

```
void T1 (void *p_arg){
               while (DEF_ON) {
                   Wait for event to occur;
                   Acquire M1;
                   Access  R1;
                   Acquire M2;
                   Access  R2;
               }
          }


          void  T2 (void *p_arg)
          {
              while (DEF_ON) {
                  Wait for event to occur;
                  Acquire M1;
                  Access  R1;
                  Acquire M2;
                  Access  R2;
              }
          }
```

Listing - Deadlock avoidance – acquire in the same order

Copyright 2015 Micrium Inc.

Synchronization

This chapter focuses on how tasks can synchronize their activities with Interrupt Service Routines (ISRs), or other tasks.

When an ISR executes, it can signal a task telling the task that an event of interest has occurred. After signaling the task, the ISR exits and, depending on the signaled task priority, the scheduler is run. The signaled task may then service the interrupting device, or otherwise react to the event. Servicing interrupting devices from task level is preferred whenever possible, since it reduces the amount of time that interrupts are disabled and the code is easier to debug.

There are two basic mechanisms for synchronizations in µC/OS-III: semaphores and event flags.

Copyright 2015 Micrium Inc.

Semaphores

About Semaphores

As defined in Resource Management, a semaphore is a protocol mechanism offered by most multitasking kernels. Semaphores were originally used to control access to shared resources. However, better mechanisms exist to protect access to shared resources. Semaphores are best used to synchronize an ISR to a task, or synchronize a task with another task as shown in the figure below.

Note that the semaphore is drawn as a flag to indicate that it is used to signal the occurrence of an event. The initial value for the semaphore is typically zero (0), indicating the event has not yet occurred.

The value “N” next to the flag indicates that the semaphore can accumulate events or credits. An ISR (or a task) can post (or signal) multiple times to a semaphore and the semaphore will remember how many times it was posted. It is possible to initialize the semaphore with a value other than zero, indicating that the semaphore initially contains that number of events.

Also, the small hourglass close to the receiving task indicates that the task has an option to specify a timeout. This timeout indicates that the task is willing to wait for the semaphore to be signaled (or posted to) within a certain amount of time. If the semaphore is not signaled within that time, µC/OS-III resumes the task and returns an error code indicating that the task was made ready-to-run because of a timeout and not the semaphore was signaled.

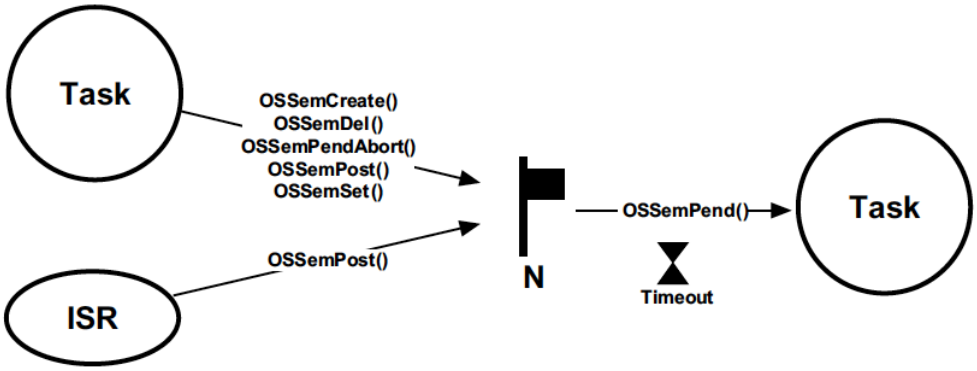


Figure - µC/OS-III Semaphore Services

There are a number of operations to perform on semaphores as summarized in the table below and the figure above. However, in this chapter, we will only discuss the three functions used most often: `OSSemCreate()`, `OSSemPend()`, and `OSSemPost()`. The other functions are described in uC-OS-III API Reference. Also note that every semaphore function is callable from a task, but only `OSSemPost()` can be called by an ISR.

Function Name	Operation
`OSSemCreate()`	Create a semaphore.
`OSSemDel()`	Delete a semaphore.
`OSSemPend()`	Wait on a semaphore.
`OSSemPendAbort()`	Abort the wait on a semaphore.
`OSSemPost()`	Signal a semaphore.
`OSSemSet()`	Force the semaphore count to a desired value.

Table - Semaphore API summary

When used for synchronization, a semaphore keeps track of how many times it was signaled using a counter. The counter can take values between 0 and 255, 65,535, or 4,294,967,295, depending on whether the semaphore mechanism is implemented using 8, 16, or 32 bits, respectively. For µC/OS-III, the maximum value of a semaphore is determined by the data type `OS_SEM_CTR` (see `os_type.h`), which is changeable, as needed (assuming access to µC/OS-III's source code). Along with the semaphore's value, µC/OS-III keeps track of tasks waiting for the semaphore to be signaled.

Unilateral Rendez-Vous

The figure below shows that a task can be synchronized with an ISR (or another task) by using a semaphore. In this case, no data is exchanged, however there is an indication that the ISR or the task (on the left) has occurred. Using a semaphore for this type of synchronization is called a unilateral rendez-vous.

Copyright 2015 Micrium Inc.

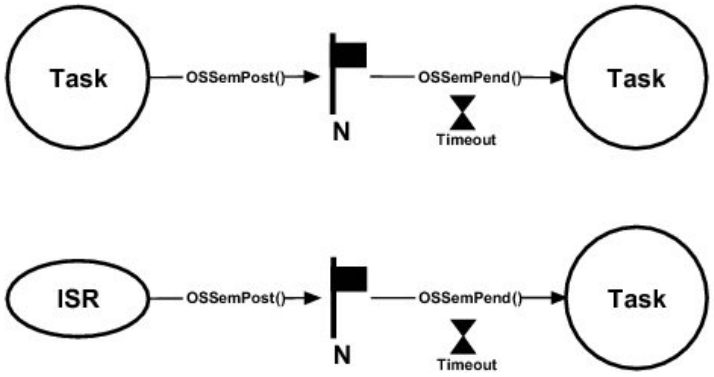


Figure - Unilateral Rendezvous

A unilateral rendez-vous is used when a task initiates an I/O operation and waits (i.e., call `OSSemPend()`) for the semaphore to be signaled (posted). When the I/O operation is complete, an ISR (or another task) signals the semaphore (i.e., calls `OSSemPost()`), and the task is resumed. This process is also shown on the timeline of the figure below and described below. The code for the ISR and task is shown in the listing follows the figure below.

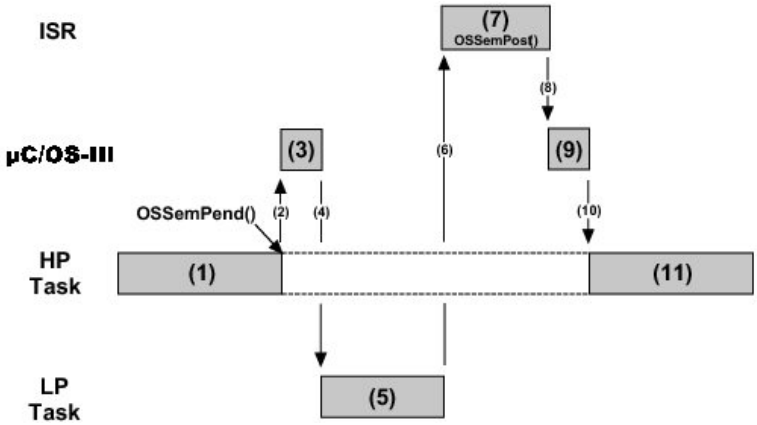


Figure - Unilateral Rendezvous, Timing Diagram

(1) A high priority task is executing. The task needs to synchronize with an ISR (i.e., wait for the ISR to occur) and call `OSSemPend()`.

(2)

(3) Since the ISR has not occurred, the task will be placed in the waiting list for the semaphore until the event occurs The scheduler in µC/OS-III will then select the next

Copyright 2015 Micrium Inc.

most important task and context switch to that task.

(5) The low-priority task executes.

(6) The event that the original task was waiting for occurs. The lower-priority task is immediately preempted (assuming interrupts are enabled), and the CPU vectors to the interrupt handler for the event.

(7)

(8) The ISR handles the interrupting device and then calls `OSSemPost()` to signal the semaphore. When the ISR completes, µC/OS-III is called (i.e. `OSIntExit()`).

(9)

(10) µC/OS-III notices that a higher-priority task is waiting for this event to occur and context switches back to the original task.

(11) The original task resumes execution immediately after the call to `OSSemPend()`.

Copyright 2015 Micrium Inc.

```
OS_SEM  MySem;


void MyISR (void)
{
    OS_ERR  err;


    /* Clear the interrupting device */
    OSSemPost(&MySem,                    (7)
              OS_OPT_POST_1,
              &err);
    /* Check “err” */
}


void MyTask (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;
    :
    :
    while (DEF_ON) {
        OSSemPend(&MySem,                (1)
                  10,
                  OS_OPT_PEND_BLOCKING,
                  &ts,
                  &err);
        /* Check “err” */                (11)
        :
        :
    }
}
```

Listing - Pending (or waiting) on a Semaphore

A few interesting things are worth noting about this process. First, the task does not need to know about the details of what happens behind the scenes. As far as the task is concerned, it called a function (`OSSemPend()`) that will return when the event it is waiting for occurs. Second, µC/OS-III maximizes the use of the CPU by selecting the next most important task, which executes until the ISR occurs. In fact, the ISR may not occur for many milliseconds and, during that time, the CPU will work on other tasks. As far as the task that is waiting for the semaphore is concerned, it does not consume CPU time while it is waiting. Finally, the task waiting for the semaphore will execute immediately after the event occurs (assuming it is the most important task that needs to run).

Copyright 2015 Micrium Inc.

Credit Tracking

As previously mentioned, a semaphore “remembers” how many times it was signaled (or posted to). In other words, if the ISR occurs multiple times before the task waiting for the event becomes the highest-priority task, the semaphore will keep count of the number of times it was signaled. When the task becomes the highest priority ready-to-run task, it will execute without blocking as many times as there were ISRs signaled. This is called Credit Tracking and is illustrated in the figure below and described below.

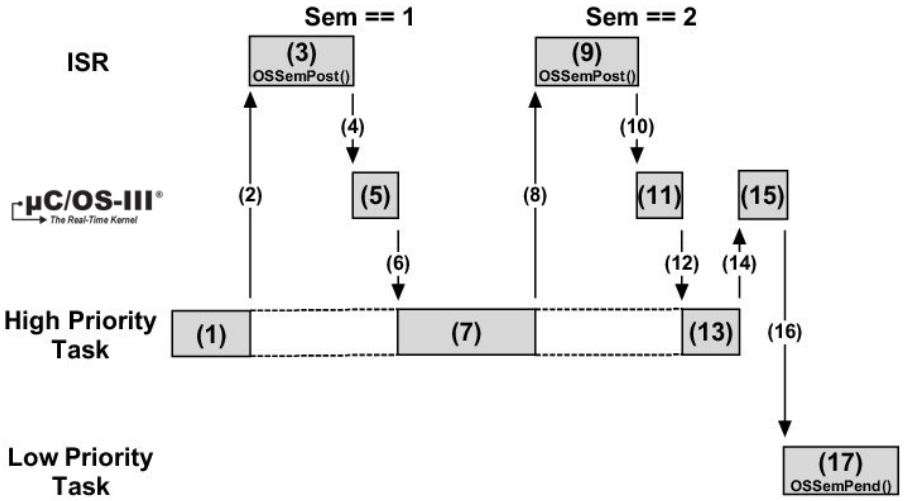


Figure - Semaphore Credit Tracking

(1) A high-priority task is executing.

(2)

(3) An event meant for a lower-priority task occurs which preempts the task (assuming interrupts are enabled). The ISR executes and posts the semaphore. At this point the semaphore count is 1.

(4)

(6) µC/OS-III is called at the end of the ISR to see if the ISR caused a higher-priority task to be ready-to-run. Since the ISR was an event that a lower-priority task was waiting for, µC/OS-III will resume execution of the higher-priority task at the exact point where it

Copyright 2015 Micrium Inc.

was interrupted.

(7) The high-priority task is resumed and continues execution.

(8)

(9) The interrupt occurs a second time. The ISR executes and posts the semaphore. At this point the semaphore count is 2.

(10)

(12) µC/OS-III is called at the end of the ISR to see if the ISR caused a higher-priority task to be ready-to-run. Since the ISR was an event that a lower-priority task was waiting for, µC/OS-III resumes execution of the higher-priority task at the exact point where it was interrupted.

(13)

(14) The high-priority task resumes execution and actually terminates the work it was doing. This task will then call one of the µC/OS-III services to wait for “its” event to occur.

(15)

(16) µC/OS-III will then select the next most important task, which happens to be the task waiting for the event and will context switch to that task.

(17) The new task executes and will know that the ISR occurred twice since the semaphore count is two. The task will handle this accordingly.

Multiple Tasks Waiting on a Semaphore

It is possible for more than one task to wait on the same semaphore, each with its own timeout as illustrated in the figure below.

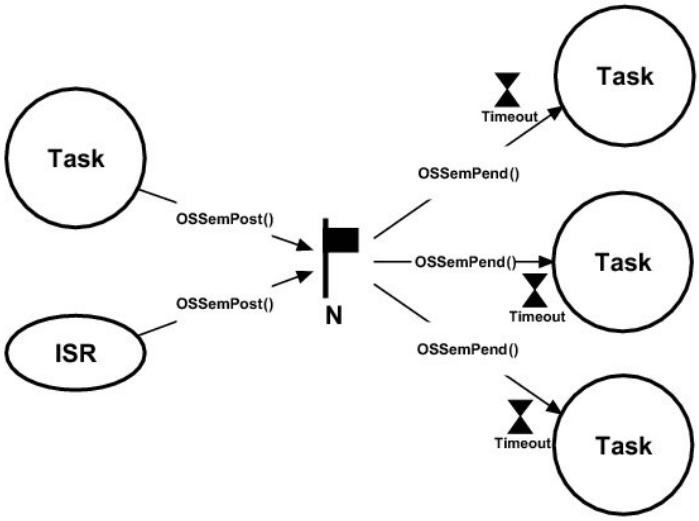


Figure - Multiple Tasks waiting on a Semaphore

When the semaphore is signaled (whether by an ISR or task), µC/OS-III makes the highest-priority task waiting on the semaphore ready-to-run. However, it is also possible to specify that all tasks waiting on the semaphore be made ready-to-run. This is called broadcasting and is accomplished by specifying `OS_OPT_POST_ALL` as an option when calling `OSSemPost()`. If any of the waiting tasks has a higher priority than the previously running task, µC/OS-III will execute the highest-priority task made ready by `OSSemPost()`.

Broadcasting is a common technique used to synchronize multiple tasks and have them start executing at the same time. However, some of the tasks that we want to synchronize might not be waiting for the semaphore. It is fairly easy to resolve this problem by combining semaphores and event flags. This will be described after examining event flags.

Semaphore Internals (for synchronization)

Note that some of the material presented in this section is also contained in Resource Management, as semaphores were also discussed in that chapter. However, the material presented here will be applicable to semaphores used for synchronization and thus will differ somewhat.

A counting semaphore allows values between 0 and 255, 65,535, or 4,294,967,295, depending on whether the semaphore mechanism is implemented using 8, 16, or 32 bits, respectively. For µC/OS-III, the maximum value of a semaphore is determined by the data type `OS_SEM_CTR` (see `os_type.h`), which can be changed as needed. Along with the semaphore's value, µC/OS-III keeps track of tasks waiting for the semaphore's availability.

Copyright 2015 Micrium Inc.

The application programmer can create an unlimited number of semaphores (limited only by available RAM). Semaphore services in µC/OS-III start with the `OSSem???()` prefix, and services available to the application programmer are described in µC-OS-III API Reference. Semaphore services are enabled at compile time by setting the configuration constant `OS_CFG_SEM_EN` to `DEF_ENABLED` in `os_cfg.h`.

Semaphores must be created before they can be used by the application. The second listing below shows how to create a semaphore.

As previously mentioned, a semaphore is a kernel object as defined by the `OS_SEM` data type, which is derived from the structure `os_sem` (see `os.h`) as shown in the listing below. The services provided by µC/OS-III to manage semaphores are implemented in the file `os_sem.c`.

```
typedef  struct  os_sem  OS_SEM;             (1)


struct  os_sem {
    OS_OBJ_TYPE          Type;               (2)
    CPU_CHAR            *NamePtr;            (3)
    OS_PEND_LIST         PendList;           (4)
    OS_SEM_CTR           Ctr;                (5)
    CPU_TS               TS;                 (6)
};
```

Listing - OS_SEM data type

(1) In µC/OS-III, all structures are given a data type. In fact, all data types start with "`OS_`" and are all uppercase. When a semaphore is declared, simply use `OS_SEM` as the data type of the variable used to declare the semaphore.

(2) The structure starts with a "`Type`" field, which allows it to be recognized by µC/OS-III as a semaphore. In other words, other kernel objects will also have a "`Type`" as the first member of the structure. If a function is passed a kernel object, µC/OS-III will confirm that it is being passed the proper data type (assuming `OS_CFG_OBJ_TYPE_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`). For example, if passing a message queue (`OS_Q`) to a semaphore service (for example `OSSemPend()`), µC/OS-III will recognize that an invalid object was passed, and return an error code accordingly.

(3) Each kernel object can be given a name to make them easier to be recognized by debuggers or µC/Probe. This member is simply a pointer to an ASCII string, which is

Copyright 2015 Micrium Inc.

assumed to be NUL terminated.

(4) Since it is possible for multiple tasks to be waiting (or pending) on a semaphore, the semaphore object contains a pend list as described in Pend Lists.

(5) A semaphore contains a counter. As explained above, the counter can be implemented as either an 8-, 16- or 32-bit value, depending on how the data type OS_SEM_CTR is declared in os_type.h. µC/OS-III keeps track of how many times the semaphore is signaled with this counter and this field is typically initialized to zero by OSSemCreate().

(6) A semaphore contains a time stamp, which is used to indicate the last time the semaphore was signaled (or posted to). µC/OS-III assumes the presence of a free-running counter that allows the application to make time measurements. When the semaphore is signaled, the free-running counter is read and the value is placed in this field, which is returned when OSSemPend() is called. This value allows the application to determine either when the signal was performed, or how long it took for the task to get control of the CPU from the signal. In the latter case, you should call OS_TS_GET() to determine the current timestamp and compute the difference.

Even for users who understand the internals of the OS_SEM data type, the application code should never access any of the fields in this data structure directly. Instead, you should always use the APIs provided with µC/OS-III.

Semaphores must be created before they can be used by an application. The listing below shows how to create a semaphore.

```
OS_SEM  MySem;                        (1)


         void  MyCode (void)
         {
             OS_ERR  err;
             :
             OSSemCreate(&MySem,               (2)
                         “My Semaphore”,       (3)
                         (OS_SEM_CTR)0,        (4)
                         &err);                (5)
             /* Check “err” */
             :
         }
```

Listing - Creating a Semaphore

(1) The application must declare a variable of type `OS_SEM`. This variable will be referenced by other semaphore services.

(2) You create a semaphore by calling `OSSemCreate()` and pass the address to the semaphore allocated in (1).

(3) You can assign an ASCII name to the semaphore, which can be used by debuggers or µC/Probe to easily identify this semaphore.

(4) You need to initialize the semaphore to zero (0) when using a semaphore as a signaling mechanism.

(5) `OSSemCreate()` returns an error code based on the outcome of the call. If all arguments are valid, `err` will contain `OS_ERR_NONE`.

(`OSSemCreate()` performs a check on the arguments passed to this function (assuming `OS_CFG_ARG_CHK_EN` is set to `DEF_ENABLED` in `os_cfg.h`) and only initializes the contents of the variable of type `OS_SEM` used for signaling.

A task waits for a signal from an ISR or another task by calling `OSSemPend()` as shown in the listing below (see Appendix A, µC-OS-III API Reference for details regarding the arguments).

```
void  MyTask (void *p_arg)
          {
              OS_ERR  err;
              CPU_TS  ts;
              :
              while (DEF_ON) {
                  OSSemPend(&MySem,                (1)
                            0,
                            OS_OPT_PEND_BLOCKING,
                            &ts,
                            &err);
              /* Check “err” */                    (2)
              :
          }
```

Listing - Pending (or waiting) on a Semaphore

(1) When called, `OSSemPend()` starts by checking the arguments passed to this function to

make sure they have valid values (assuming OS_CFG_OBJ_TYPE_CHK_EN is set to DEF_ENABLED in os_cfg.h).

If the semaphore counter (.Ctr of OS_SEM) is greater than zero, the counter is decremented and OSSemPend() returns, which indicates that the signal occurred. This is the outcome that the caller expects.

If the semaphore counter is zero, this indicates that the signal has not occurred and the calling task might need to wait for the semaphore to be released. If you specify OS_OPT_PEND_NON_BLOCKING as the option (the task is not to block), OSSemPend() returns immediately to the caller and the returned error code will indicate that the signal did not occur.

If you specify OS_OPT_PEND_BLOCKING as the option, the calling task will be inserted in the list of tasks waiting for the semaphore to be signaled. The task is inserted in the list by priority order with the highest priority task waiting on the semaphore at the beginning of the list as shown in the figure at the end of this page.

If you further specify a non-zero timeout, the task will also be inserted in the tick list. A zero value for a timeout indicates that the calling task is willing to wait forever for the semaphore to be signaled.

The scheduler is then called since the current task is not able to run (it is waiting for the semaphore to be signaled). The scheduler will then run the next highest-priority task that is ready-to-run.

When the semaphore is signaled and the task that called OSSemPend() is again the highest-priority task, a task status is examined to determine the reason why OSSemPend() is returning to its caller. The possibilities are:

1) The semaphore was signaled which is the desired outcome

2) The pend was aborted by another task

3) The semaphore was not signaled within the specified timeout

4) The semaphore was deleted

Copyright 2015 Micrium Inc.

When OSSemPend() returns, the caller is notified of the above outcome through an appropriate error code.

(2) If OSSemPend() returns with err set to OS_ERR_NONE, you can assume that the semaphore was signaled and the task can proceed with servicing the ISR or task that caused the signal. If err contains anything else, OSSemPend() either timed out (if the timeout argument was non-zero), the pend was aborted by another task, or the semaphore was deleted by another task. It is always important to examine returned error code and not assume everything went as expected.

To signal a task (either from an ISR or a task), simply call OSSemPost() as shown in the listing below.

```
OS_SEM  MySem;


          void MyISR (void)
          {
              OS_ERR  err;
              :
              OSSemPost(&MySem,                (1)
                        OS_OPT_POST_1,         (2)
                        &err);                 (3)
              /* Check “err” */
              :
              :
          }
```

Listing - Posting (or signaling) a Semaphore

(1) Your task signals (or posts to) the semaphore by calling OSSemPost(). You specify the semaphore to post by passing its address. The semaphore must have been previously created.

(2) The next argument specifies how the task wants to post. There are a number of options to choose from.

When you specify OS_OPT_POST_1, you are indicating that you want to post to only one task (in case there are multiple tasks waiting on the semaphore). The task that will be made ready-to-run will be the highest-priority task waiting on the semaphore. If there are multiple tasks at the same priority, only one of them will be made ready-to-run. As

Copyright 2015 Micrium Inc.

shown in the figure below, tasks waiting are in priority order (HPT means High Priority Task and LPT means Low Priority Task). So, it is a fast operation to extract the HPT from the list.

If specifying OS_OPT_POST_ALL, all tasks waiting on the semaphore will be posted and made ready-to-run.

The calling task can “add” the option OS_OPT_POST_NO_SCHED to either of the two previous options to indicate that the scheduler is not to be called at the end of OSSemPost(), possibly because additional postings will be performed, and rescheduling should only take place when finished. This means that the signal is performed, but the scheduler is not called even if a higher-priority task was waiting for the semaphore to be signaled. This allows the calling task to perform other post functions (if needed) and make all the posts take effect simultaneously. Note that OS_OPT_POST_NO_SCHED is “additive,” meaning that it can be used with either of the previous options. You can thus specify:

```
OS_OPT_POST_1
OS_OPT_POST_ALL
OS_OPT_POST_1   + OS_OPT_POST_NO_SCHED
OS_OPT_POST_ALL + OS_OPT_POST_NO_SCHED
```

Copyright 2015 Micrium Inc.

Task Semaphore

About Task Semaphores

Signalling a task using a semaphore is a very popular method of synchronization and, in µC/OS-III, each task has its own built-in semaphore. This feature not only simplifies code, but is also more efficient than using a separate semaphore object. The semaphore, which is built into each task, is shown in the figure below.

Task semaphore services in µC/OS-III start with the `OSTaskSem???()` prefix, and the services available to the application programmer are described in Appendix A, "µC/OS-III API Reference". Task semaphores are built into µC/OS-III and cannot be disabled at compile time as can other services. The code for task semaphores is found in `os_task.c`.

You can use this feature if your code knows which task to signal when the event occurs. For example, if you receive an interrupt from an Ethernet controller, you can signal the task responsible for processing the received packet as it is preferable to perform this processing using a task instead of the ISR.

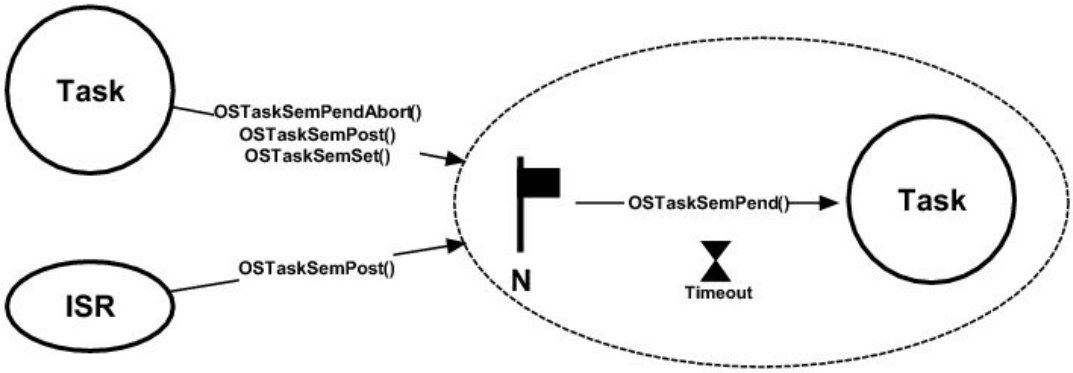


Figure - Semaphore built-into a Task

There is a variety of operations to perform on task semaphores, summarized in the table below.

Copyright 2015 Micrium Inc.

Function Name	Operation
OSTaskSemPend()	Wait on a task semaphore.
OSTaskSemPendAbort()	Abort the wait on a task semaphore.
OSTaskSemPost()	Signal a task.
OSTaskSemSet()	Force the semaphore count to a desired value.

Table - Task Semaphore API summary

Pending (i.e., Waiting) on a Task Semaphore

When a task is created, it automatically creates an internal semaphore with an initial value of zero (0). Waiting on a task semaphore is quite simple, as shown in the listing below .

```
void MyTask (void *p_arg)
{
    OS_ERR err;
    CPU_TS ts;
    :
    while (DEF_ON) {
        OSTaskSemPend(10,                   (1)
                      OS_OPT_PEND_BLOCKING, (2)
                      &ts,                  (3)
                      &err);                (4)
        /* Check “err” */
    :
    :
 }
```

Listing - Pending (or waiting) on Task’s internal semaphore

(1) A task pends (or waits) on the task semaphore by calling `OSTaskSemPend()`. There is no need to specify which task, as the current task is assumed. The first argument is a timeout specified in number of clock ticks. The actual timeout obviously depends on the tick rate. If the tick rate (see `os_cfg_app.h`) is set to 1000, a timeout of 10 ticks represents 10 milliseconds. Specifying a timeout of zero (0) means that the task will wait forever for the task semaphore.

(2) The second argument specifies how to pend. There are two options: `OS_OPT_PEND_BLOCKING` and `OS_OPT_PEND_NON_BLOCKING`. The blocking option means that, if the task semaphore has not been signaled (or posted to), the task will wait until the semaphore is signaled, the pend is aborted by another task or, until the timeout expires.

Copyright 2015 Micrium Inc.

(3) When the semaphore is signaled, µC/OS-III reads a “timestamp” and places it in the receiving task’s `OS_TCB`. When `OSTaskSemPend()` returns, the value of the timestamp is placed in the local variable “`ts`”. This feature captures “when” the signal actually happened. You can call `OS_TS_GET()` to read the current timestamp and compute the difference. This establishes how long it took for the task to receive the signal from the posting task or ISR.

(4) `OSTaskSemPend()` returns an error code based on the outcome of the call. If the call was successful, err will contain `OS_ERR_NONE`. If not, the error code will indicate the reason of the error (see µC-OS-III API Reference for a list of possible error code for `OSTaskSemPend()`.

Posting (i.e., Signaling) a Task Semaphore

An ISR or a task signals a task by calling OSTaskSemPost(), as shown in the listing below.

```
OS_TCB MyTaskTCB;


void MyISR (void *p_arg)
{
    OS_ERR err;
    :
    OSTaskSemPost(&MyTaskTCB, (1)
                  OS_OPT_POST_NONE, (2)
                  &err); (3)
    /* Check “err” */
    :
    :
}
```

Listing - Posting (or signaling) a Semaphore

(1) A task posts (or signals) the task by calling `OSTaskSemPost()`. It is necessary to pass the address of the desired task’s `OS_TCB` and of course, the task must exist.

(2) The next argument specifies how the user wants to post. There are only two choices.

Specify `OS_OPT_POST_NONE`, which indicates the use of the default option of calling the scheduler after posting the semaphore.

Copyright 2015 Micrium Inc.

Or, specify `OS_OPT_POST_NO_SCHED` to indicate that the scheduler is not to be called at the end of `OSTaskSemPost()`, possibly because there will be additional postings, and rescheduling would take place when finished (the last post would not specify this option).

(3) `OSTaskSemPost()` returns an error code based on the outcome of the call. If the call was successful, err will contain `OS_ERR_NONE`. If not, the error code will indicate the reason of the error (see uC-OS-III API Reference for a list of possible error codes for `OSTaskSemPost()`.

Bilateral Rendez-Vous – Task Synchronization

Two tasks can synchronize their activities by using two task semaphores, as shown in the figure below, and is called a *bilateral rendez-vous*. A bilateral rendez-vous is similar to a unilateral rendez-vous, except that both tasks must synchronize with one another before proceeding. A bilateral rendez-vous cannot be performed between a task and an ISR because an ISR cannot wait on a semaphore.

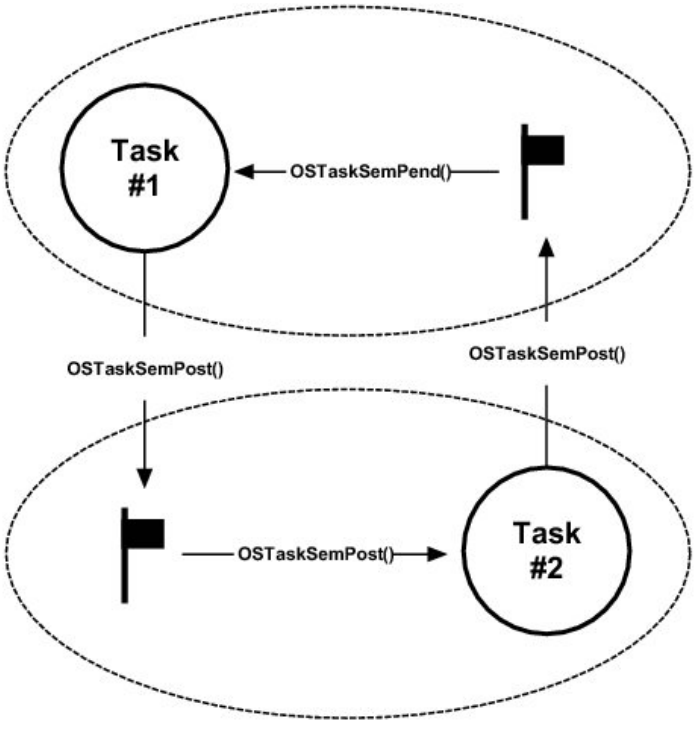


Figure - Bilateral Rendezvous

The code for a bilateral rendez-vous is shown in the listing below. Of course, a bilateral rendez-vous can use two separate semaphores, but the built-in task semaphore makes setting up this type of synchronization quite straightforward.

```
OS_TCB  MyTask1_TCB;
OS_TCB  MyTask2_TCB;


void Task1 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;

    while (DEF_ON) {
        :
        OSTaskSemPost(&MyTask2_TCB,             (1)
                      OS_OPT_POST_NONE,
                      &err);
        /* Check ’err” */
        OSTaskSemPend(0,                        (2)
                      OS_OPT_PEND_BLOCKING,
                      &ts,
                      &err);
        /* Check ’err” */
        :
    }
}


void Task2 (void *p_arg)
{
    OS_ERR  err;
    CPU_TS  ts;

    while (DEF_ON) {
        :
        OSTaskSemPost(&MyTask1_TCB,             (3)
                      OS_OPT_POST_NONE,
                      &err);
        /* Check ’err” */
        OSTaskSemPend(0,                        (4)
                      OS_OPT_PEND_BLOCKING,
                      &ts,
                      &err);
        /* Check ’err” */
        :
    }
}
```

Listing - Tasks synchronizing their activities

(1) Task #1 is executing and signals Task #2’s semaphore.

(2) Task #1 pends on its internal semaphore to synchronize with Task #2. Because Task #2 has not executed yet, Task #1 is blocked waiting on its semaphore to be signaled.

Copyright 2015 Micrium Inc.

µC/OS-III context switches to Task #2.

(3) Task #2 executes, and signals Task #1’s semaphore.

(4) Since it has already been signaled, Task #2 is now synchronized to Task #1. If Task #1 is higher in priority than Task #2, µC/OS-III will switch back to Task #1. If not, Task #2 continues execution.

Copyright 2015 Micrium Inc.

Event Flags

About Event Flags

Event flags are used when a task needs to synchronize with the occurrence of multiple events. The task can be synchronized when any of the events have occurred, which is called disjunctive synchronization (logical OR). A task can also be synchronized when all events have occurred, which is called conjunctive synchronization (logical AND). Disjunctive and conjunctive synchronization are shown in the figure below.

The application programmer can create an unlimited number of event flag groups (limited only by available RAM). Event flag services in C/OS-III start with the `OSFlag???()` prefix. The services available to the application programmer are described in µC-OS-III API Reference.

The code for event flag services is found in the file `os_flag.c`, and is enabled at compile time by setting the configuration constant `OS_CFG_FLAG_EN` to `DEF_ENABLED` in `os_cfg.h`.

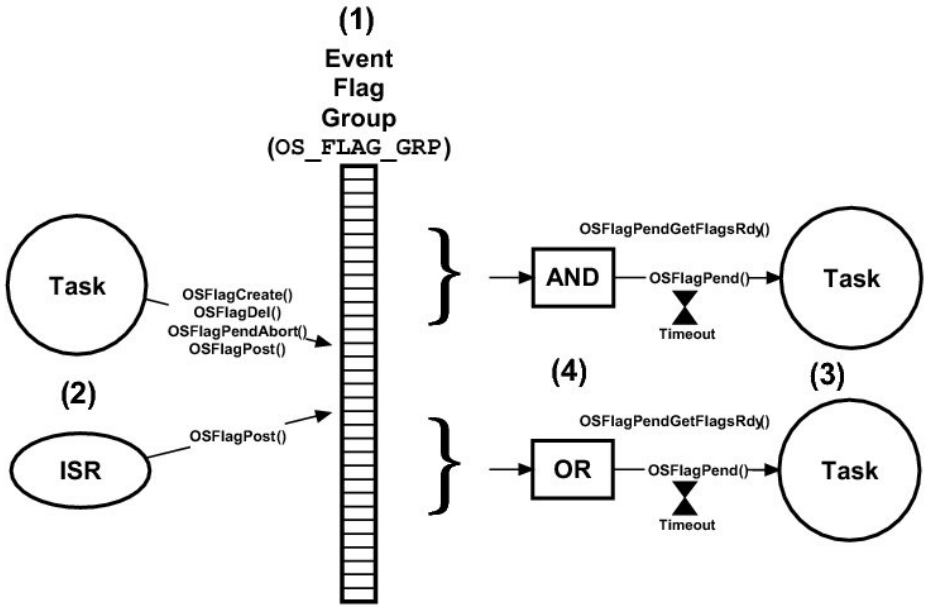


Figure - Event Flags

(1) A C/OS-III "event flag group" is a kernel object of type `OS_FLAG_GRP` (see `os.h`), and

consists of a series of bits (8-, 16- or 32-bits, based on the data type `OS_FLAGS` defined in `os_type.h`). The event flag group also contains a list of tasks waiting for some (or all) of the bits to be set or clear – 1 or 0. An event flag group must be created before it can be used by tasks and ISRs. You need to create event flags prior to starting C/OS-III, or by a startup task in the application code.

(2) Tasks or ISRs can post to event flags. In addition, only tasks can create, delete, and stop other task from pending on event flag groups.

(3) A task can wait (i.e., pend) on any number of bits in an event flag group (i.e., a subset of all the bits). As with all C/OS-III pend calls, the calling task can specify a timeout value such that if the desired bits are not posted within a specified amount of time (in ticks), the pending task is resumed and informed about the timeout.

(4) The task can specify whether it wants to wait for “any” subset of bits (OR) to be set (or clear), or wait for “all” bits in a subset of bit (AND) to be set (or clear).

There are a number of operations to perform on event flags, as summarized in the table below.

Function	Name Operation
`OSFlagCreate()`	Create an event flag group
`OSFlagDel()`	Delete an event flag group
`OSFlagPend()`	Pend (i.e., wait) on an event flag group
`OSFlagPendAbort()`	Abort waiting on an event flag group
`OSFlagPendGetFlagsRdy()`	Get flags that caused task to become ready
`OSFlagPost()`	Post flag(s) to an event flag group

Table - Event Flags API summary

Event Flags Internals

The application programmer can create an unlimited number of event flag groups (limited only by available RAM). Event flag services in µC/OS-III start with `OSFlag` and the services available to the application programmer are described in the topics below:

- OSFlagCreate

- OSFlagDel
- OSFlagPend
- OSFlagPendAbort
- OSFlagPendGetFlagsRdy
- OSFlagPost

Event flag services are enabled at compile time by setting the configuration constant `OS_CFG_FLAG_EN` to `DEF_ENABLED` in `os_cfg.h`.

An event flag group is a kernel object as defined by the `OS_FLAG_GRP` data type, which is derived from the structure `os_flag_grp` (see `os.h`) as shown in the listing below.

The services provided by µC/OS-III to manage event flags are implemented in the file `os_flag.c`.

```
typedef  struct  os_flag_grp  OS_FLAG_GRP;   (1)


struct  os_flag_grp {
    OS_OBJ_TYPE          Type;              (2)
    CPU_CHAR            *NamePtr;           (3)
    OS_PEND_LIST         PendList;          (4)
    OS_FLAGS             Flags;             (5)
    CPU_TS               TS;                (6)
};
```

Listing - OS_FLAG_GRP data type

(1) In µC/OS-III, all structures are given a data type. In fact, all data types start with "`OS_`" and are uppercase. When an event flag group is declared, you simply use `OS_FLAG_GRP` as the data type of the variable used to declare the event flag group.

(2) The structure starts with a "`Type`" field, which allows it to be recognized by µC/OS-III as an event flag group. In other words, other kernel objects will also have a "`Type`" as the first member of the structure. If a function is passed a kernel object, µC/OS-III will be able to confirm that it is being passed the proper data type (assuming

Copyright 2015 Micrium Inc.

OS_CFG_OBJ_TYPE_CHK_EN is set to 1 in os_cfg.h). For example, if passing a message queue (OS_Q) to an event flag service (for example OSFlagPend()), µC/OS-III will be able to recognize that an invalid object was passed, and return an error code accordingly.

(3) Each kernel object can be given a name to make them easier to be recognized by debuggers or µC/Probe. This member is simply a pointer to an ASCII string, which is assumed to be NUL terminated.

(4) Because it is possible for multiple tasks to be waiting (or pending) on an event flag group, the event flag group object contains a pend list as described in Pend Lists.

(5) An event flag group contains a series of flags (i.e., bits), and this member contains the current state of these flags. The flags can be implemented using either an 8-, 16- or 32-bit value depending on how the data type OS_FLAGS is declared in os_type.h.

(6) An event flag group contains a timestamp used to indicate the last time the event flag group was posted to. µC/OS-III assumes the presence of a free-running counter that allows users to make time measurements. When the event flag group is posted to, the free-running counter is read and the value is placed in this field, which is returned when OSFlagPend() is called. This value allows an application to determine either when the post was performed, or how long it took for your code to obtain control of the CPU from the post. In the latter case, you can call OS_TS_GET() to determine the current timestamp and compute the difference.

Even if the user understands the internals of the OS_FLAG_GRP data type, application code should never access any of the fields in this data structure directly. Instead, you should always use the APIs provided with µC/OS-III.

Event flag groups must be created before they can be used by an application as shown in the listing below.

Copyright 2015 Micrium Inc.

```
OS_FLAG_GRP  MyEventFlagGrp;                (1)


void  MyCode (void)
{
    OS_ERR  err;
    :
    OSFlagCreate(&MyEventFlagGrp,           (2)
                 “My Event Flag Group”,     (3)
                 (OS_FLAGS)0,               (4)
                 &err);                     (5)
    /* Check ’err” */
    :
}
```

Listing - Creating a Event Flag Group

(1) The application must declare a variable of type `OS_FLAG_GRP`. This variable will be referenced by other event flag services.

(2) You create an event flag group by calling `OSFlagCreate()` and pass the address to the event flag group allocated in (1).

(3) You can assign an ASCII name to the event flag group, which can be used by debuggers or µC/Probe to easily identify this event flag group. µC/OS-III stores a pointer to the name so there is no practical limit to its size, except that the ASCII string needs to be `NUL` terminated.

(4) You initialize the flags inside the event flag group to zero (0) unless the task and ISRs signal events with bits cleared instead of bits set. If using cleared bits, you should initialize all the bits to ones (1).

(5) `OSFlagCreate()` returns an error code based on the outcome of the call. If all the arguments are valid, `err` will contain `OS_ERR_NONE`.

A task waits for one or more event flag bits either from an ISR or another task by calling `OSFlagPend()` as shown in the listing below.

Copyright 2015 Micrium Inc.

```
OS_FLAG_GRP  MyEventFlagGrp;


void MyTask (void *p_arg)
{
    OS_ERR   err;
    CPU_TS   ts;
    OS_FLAGS which_flags;
    :
    while (DEF_ON) {
        :
        which_flags = OSFlagPend(&MyEventFlagGrp,   /* (1) Pointer to event flag group        */
                                 (OS_FLAGS)0x0F,    /*     Which bits to wait on              */
                                 10,                /*     Maximum time to wait               */
                                 OS_OPT_PEND_BLOCKING +
                                 OS_OPT_PEND_FLAG_SET_ANY, /*     Option(s)                  */
                                 &ts,               /*     Timestamp of when posted to */
                                 &err);             /*     Pointer to Error returned   */
        /* Check “err”                              (2)                                    */
        :
        :
    }
}
```

Listing - Pending (or waiting) on an Event Flag Group

(1) When called, `OSFlagPend()` starts by checking the arguments passed to this function to ensure they have valid values (assuming `OS_CFG_ARG_CHK_EN` is set to `1` in `os_cfg.h`). If the bits the task is waiting for are set (or cleared depending on the option), `OSFlagPend()` returns and indicate which flags satisfied the condition. This is the outcome that the caller expects.

If the event flag group does not contain the flags that the caller is looking for, the calling task might need to wait for the desired flags to be set (or cleared). If you specify `OS_OPT_PEND_NON_BLOCKING` as the option (the task is not to block), `OSFlagPend()` returns immediately to the caller and the returned error code indicates that the bits have not been set (or cleared).

If you specify `OS_OPT_PEND_BLOCKING` as the option, the calling task will be inserted in the list of tasks waiting for the desired event flag bits. The task is not inserted in priority order but simply inserted at the beginning of the list. This is done because whenever bits are set (or cleared), it is necessary to examine all tasks in this list to see if their desired bits have been satisfied.

If you further specify a non-zero `timeout`, the task will also be inserted in the tick list. A zero value for a `timeout` indicates that the calling task is willing to wait forever for the

Copyright 2015 Micrium Inc.

desired bits.

The scheduler is then called since the current task is no longer able to run (it is waiting for the desired bits). The scheduler will run the next highest-priority task that is ready-to-run.

When the event flag group is posted to and the task that called `OSFlagPend()` has its desired bits set or cleared, a task status is examined to determine the reason why `OSFlagPend()` is returning to its caller. The possibilities are:

1. The desired bits were set (or cleared)
2. The pend was aborted by another task
3. The bits were not set (or cleared) within the specified timeout
4. The event flag group was deleted

When `OSFlagPend()` returns, the caller is notified of the above outcome through an appropriate error code.

(2) If `OSFlagPend()` returns with `err` set to `OS_ERR_NONE`, you can assume that the desired bits were set (or cleared) and the task can proceed with servicing the ISR or task that created those events. If `err` contains anything else, `OSFlagPend()` either timed out (if the timeout argument was non-zero), the pend was aborted by another task or, the event flag group was deleted by another task. It is always important to examine the returned error code and not assume everything went as planned.

To set (or clear) event flags (either from an ISR or a task), you simply call `OSFlagPost()` , as shown in the listing below:

Copyright 2015 Micrium Inc.

```
OS_FLAG_GRP  MyEventFlagGrp;

void MyISR (void)
{
    OS_ERR    err;
    OS_FLAGS  flags_cur;
    :
    flags_cur = OSFlagPost(&MyEventFlagGrp,       (1)
                           (OS_FLAGS)0x0C,        (2)
                           OS_OPT_POST_FLAG_SET,  (3)
                           &err);                 (4)
    /* Check ’err” */
    :
    :
}
```

Listing - Posting flags to an Event Flag Group

(1) A task posts to the event flag group by calling `OSFlagPost()`. Specify the desired event flag group to post by passing its address. Of course, the event flag group must have been previously created. `OSFlagPost()` returns the current value of the event flags in the event flag group after the post has been performed.

(2) The next argument specifies which bit(s) the ISR (or task) will be setting or clearing in the event flag group.

(3) You can specify `OS_OPT_POST_FLAG_SET` or `OS_OPT_POST_FLAG_CLR`.

If you specify `OS_OPT_POST_FLAG_SET`, the bits specified in the second arguments will set the corresponding bits in the event flag group. For example, if `MyEventFlagGrp.Flags` contains 0x03, the code in the listing will change `MyEventFlagGrp.Flags` to `0x0F`.

If you specify `OS_OPT_POST_FLAG_CLR`, the bits specified in the second arguments will clear the corresponding bits in the event flag group. For example, if `MyEventFlagGrp.Flags` contains `0x0F`, the code in the listing will change `MyEventFlagGrp.Flags` to `0x03`.

When calling `OSFlagPost()` you can specify as an option (i.e., `OS_OPT_POST_NO_SCHED`) to not call the scheduler. This means that the post is performed, but the scheduler is not called even if a higher-priority task was waiting for the event flag group. This allows the calling task to perform other post functions (if needed) and make all the posts take effect simultaneously.

Copyright 2015 Micrium Inc.

(4) `OSFlagPost()` returns an error code based on the outcome of the call. If the call was successful, `err` will contain `OS_ERR_NONE`. If not, the error code will indicate the reason of the error (see Appendix A, µC-OS-III Configuration Manual for a list of possible error codes for `OSFlagPost()`.

Using Event Flags

When a task or an ISR posts to an event flag group, all tasks that have their wait conditions satisfied will be resumed.

It’s up to the application to determine what each bit in an event flag group means and it is possible to use as many event flag groups as needed. In an event flag group you can, for example, define that bit #0 indicates that a temperature sensor is too low, bit #1 may indicate a low battery voltage, bit #2 could indicate that a switch was pressed, etc. The code (tasks or ISRs) that detects these conditions would set the appropriate event flag by calling `OSFlagPost()` and the task(s) that would respond to those conditions would call `OSFlagPend()`.

The listing below shows how to use event flags.

Copyright 2015 Micrium Inc.

```
#define      TEMP_LOW   (OS_FLAGS)0x0001                          (1)
#define      BATT_LOW   (OS_FLAGS)0x0002
#define      SW_PRESSED (OS_FLAGS)0x0004

OS_FLAG_GRP  MyEventFlagGrp;                                      (2)

void main (void)
{
    OS_ERR  err;

    OSInit(&err);
    :
    OSFlagCreate(&MyEventFlagGrp,                                 (3)
                 ”My Event Flag Group”,
                 (OS_FLAGS)0,
                 &err);
    /* Check ’err” */
    :
    OSStart(&err);
}


void  MyTask (void *p_arg)                                        (4)
{
    OS_ERR  err;
    CPU_TS  ts;

    while (DEF_ON) {
        OSFlagPend(&MyEventFlagGrp,                               (5)
                   TEMP_LOW + BATT_LOW,
                   (OS_TICK )0,
                   (OS_OPT)OS_OPT_PEND_FLAG_SET_ANY,
                   &ts,
                   &err);
        /* Check ’err” */
        :
    }
}
void  MyISR (void)                                                (6)
{
    OS_ERR  err;
    :
    OSFlagPost(&MyEventFlagGrp,                                   (7)
               BAT_LOW,
               (OS_OPT)OS_OPT_POST_FLAG_SET,
               &err);
    /* Check ’err” */
    :
}
```

Listing - Using Event Flags

(1) You need to define some bits in the event flag group.

(2) You have to declare an object of type `OS_FLAG_GRP`. This object will be referenced in all subsequent µC/OS-III calls that apply to this event flag group. For the sake of discussions, assume that event flags are declared to be 16-bits in `os_type.h` (i.e., of type

Copyright 2015 Micrium Inc.

CPU_INT16U).

(3) Event flag groups must be “created” before they can be used. The best place to do this is in your startup code as it ensures that no tasks, or ISR, will be able to use the event flag group until µC/OS-III is started. In other words, the best place is to create the event flag group is in main(). In the example, the event flag was given a name and all bits start in their cleared state (i.e., all zeros).

(4) You can assume here that the application created “MyTask()” which will be pending on the event flag group.

(5) To pend on an event flag group, you call OSFlagPend() and pass it the address of the desired event flag group.

The second argument specifies which bits the task will be waiting to be set (assuming the task is triggered by set bits instead of cleared bits).

You also need to specify how long to wait for these bits to be set. A timeout value of zero (0) indicates that the task will wait forever. A non-zero value indicates the number of ticks the task will wait until it is resumed if the desired bits are not set.

Specifying OS_OPT_FLAG_SET_ANY indicates that the task will wake up if either of the two bits specified is set.

A timestamp is read and saved when the event flag group is posted to. This timestamp can be used to determine the response time to the event.

OSFlagPend() performs a number of checks on the arguments passed (i.e., did you pass NULL pointers, invalid options, etc.), and returns an error code based on the outcome of the call (assuming OS_CFG_ARG_CHK_EN is set to DEF_ENABLED in os_cfg.h). If the call was successful “err” will be set to OS_ERR_NONE.

(6) An ISR (it can also be a task) is setup to detect when the battery voltage of the product goes low (assuming the product is battery operated). The ISR signals the task, letting the task perform whatever corrective action is needed.

(7) The desired event flag group is specified in the post call as well as which flag the ISR is setting. The third option specifies that the error condition will be “flagged” as a set bit.

Copyright 2015 Micrium Inc.

Again, the function sets “err” based on the outcome of the call.

Event flags are generally used for two purposes: status and transient events. Typically you would use different event flag groups to handle each of these as shown in the figure below .

Tasks or ISRs can report status information such as a temperature that has exceeded a certain value, that RPM is zero on an engine or motor, or there is fuel in the tank, and more. This status information cannot be “consumed” by the tasks waiting for these events, because the status is managed by other tasks or ISRs. Event flags associated with status information are monitored by other task by using non-blocking wait calls.

Tasks will report transient events such as a switch was pressed, an object was detected by a motion sensor, an explosion occurred, etc. The task that responds to these events will typically block waiting for any of those events to occur and “consume” the event.

Copyright 2015 Micrium Inc.

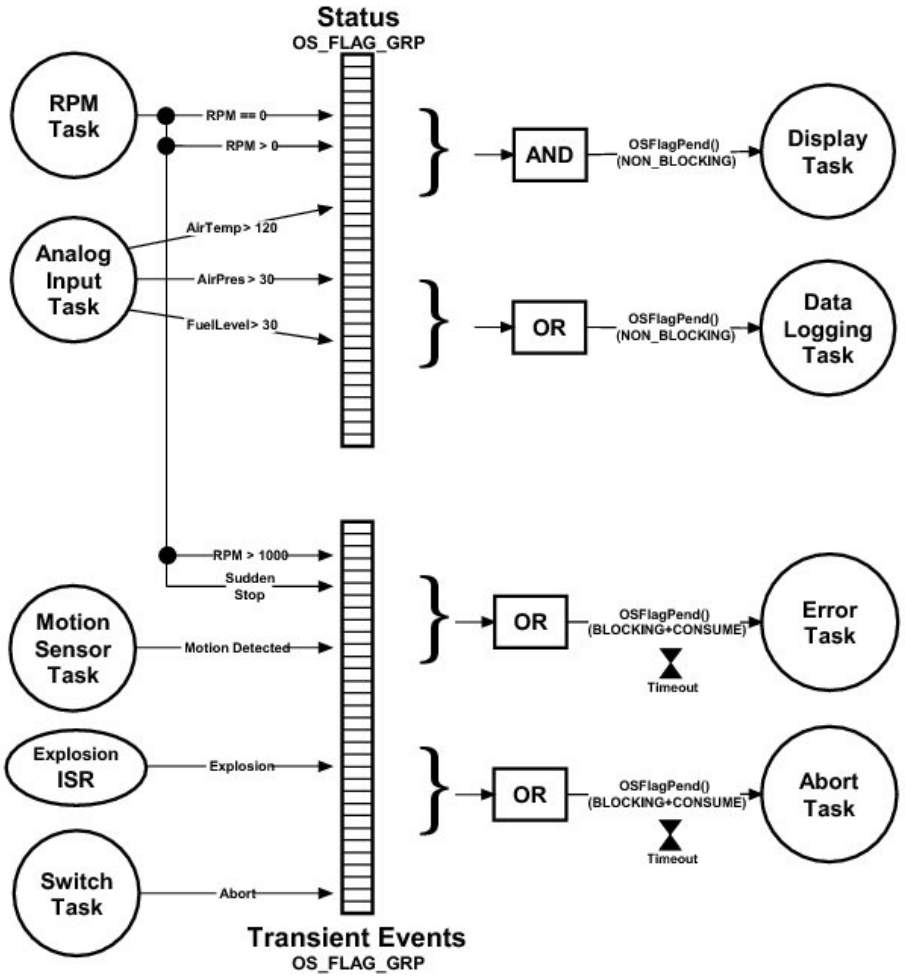


Figure - Event Flags used for Status and Transient Events

Synchronizing Multiple Tasks

Synchronizing the execution of multiple tasks by broadcasting to a semaphore is a commonly used technique. It may be important to have multiple tasks start executing at the same time. Obviously, on a single processor, only one task will actually execute at one time. However, the start of their execution will be synchronized to the same time. This is called a multiple task rendez-vous. However, some of the tasks synchronized might not be waiting for the semaphore when the broadcast is performed. It is fairly easy to resolve this problem by combining semaphores and event flags, as shown in the figure below For this to work properly, the task on the left needs to have a lower priority than the tasks waiting on the semaphore.

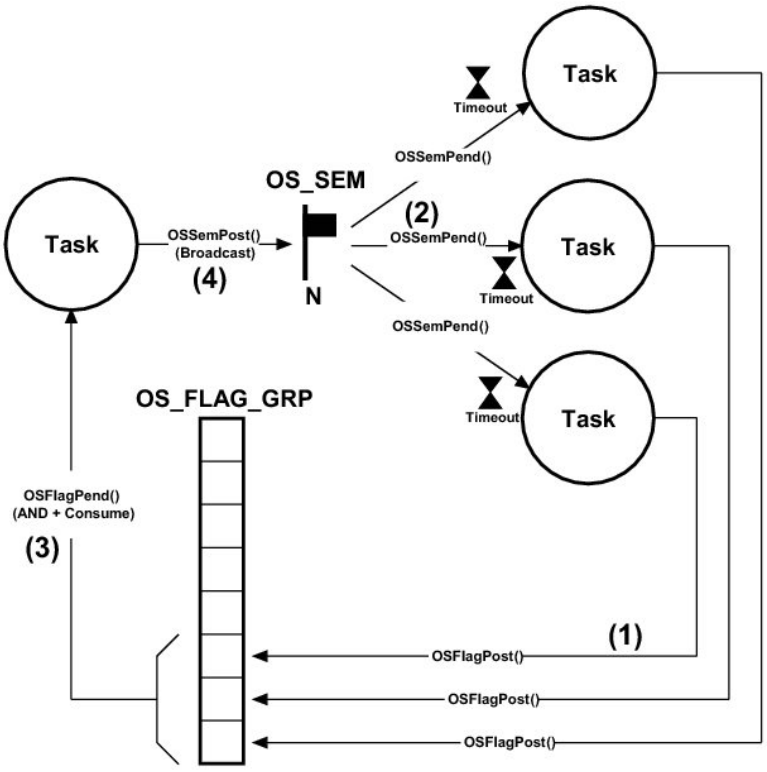


Figure - Multiple Task Rendezvous

(1) Each task that needs to synchronize at the rendez-vous needs to set an event flag bit (and specify `OS_OPT_POST_NO_SCHED`).

(2) The task needs to wait for the semaphore to be signaled.

(3) The task that will be broadcasting must wait for "`all`" of the event flags corresponding to each task to be set.

(4) When all waiting tasks are ready, the task that will synchronize the waiting task issues a broadcast to the semaphore.

Copyright 2015 Micrium Inc.

Message Passing

It is sometimes necessary for a task or an ISR to communicate information to another task. This information transfer is called *inter-task* communication. Information can be communicated between tasks in two ways: through global data, or by sending messages.

As seen in Resource Management, when using global variables, each task or ISR must ensure that it has exclusive access to variables. If an ISR is involved, the only way to ensure exclusive access to common variables is to disable interrupts. If two tasks share data, each can gain exclusive access to variables either by disabling interrupts, locking the scheduler, using a semaphore, or preferably, using a mutual-exclusion semaphore. Note that a task can only communicate information to an ISR by using global variables. A task is not aware when a global variable is changed by an ISR, unless the ISR signals the task, or the task polls the contents of a variable periodically.

Messages can either be sent to an intermediate object called a *message queue*, or directly to a task since in µC/OS-III, each task has its own built-in message queue. You can use an external message queue if multiple tasks are to wait for messages. You would send a message directly to a task if only one task will process the data received.

When a task waits for a message to arrive, it does not consume CPU time.

Copyright 2015 Micrium Inc.

Messages

A message consists of a pointer to data, a variable containing the size of the data being pointed to, and a timestamp indicating when the message was sent. The pointer can point to a data area or even a function. Obviously, the sender and the receiver must agree as to the contents and the meaning of the message. In other words, the receiver of the message will need to know the meaning of the message received to be able to process it. For example, an Ethernet controller receives a packet and sends a pointer to this packet to a task that knows how to handle the packet.

The message contents must always remain in scope since the data is actually sent by reference instead of by value. In other words, data sent is not copied. You might consider using dynamically allocated memory as described in Memory Management API Changes. Alternatively, you can pass a pointer to a global variable, a global data structure, a global array, or a function, etc.

Copyright 2015 Micrium Inc.

Message Queues

A message queue is a kernel object allocated by the application. In fact, you can allocate any number of message queues. The only limit is the amount of RAM available.

There are a number of operations that the user can perform on message queues, summarized in the figure below. However, an ISR can only call `OSQPost()`. A message queue must be created before sending messages through it.

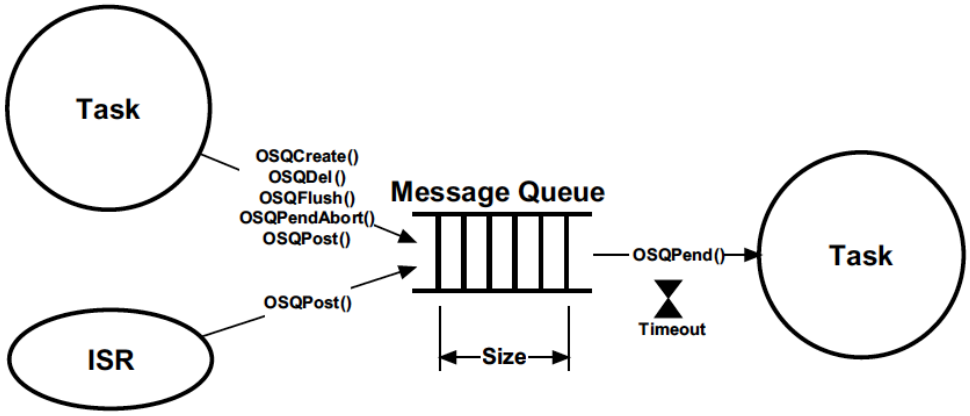


Figure - Operations on message queue

Message queues are drawn as a first-in, first-out pipe (FIFO). However, with C/OS-III, it is possible to post messages in last-in, first-out order (LIFO). The LIFO mechanism is useful when a task or an ISR must send an “urgent” message to a task. In this case, the message bypasses all other messages already in the message queue. The size of the message queue is configurable at run-time.

The small hourglass close to the receiving task (in the figure above) indicates that the task has an option to specify a timeout. This timeout indicates that the task is willing to wait for a message to be sent to the message queue within a certain amount of time. If the message is not sent within that time, C/OS-III resumes the task and returns an error code indicating that the task was made ready-to-run because of a timeout, and not because the message was received. It is possible to specify an infinite timeout and indicate that the task is willing to wait forever for the message to arrive.

The message queue also contains a list of tasks waiting for messages to be sent to the message queue. Multiple tasks can wait on a message queue as shown in the figure below. When a message is sent to the message queue, the highest priority task waiting on the message queue

Copyright 2015 Micrium Inc.

receives the message. Optionally, the sender can *broadcast* a message to all tasks waiting on the message queue. In this case, if any of the tasks receiving the message from the broadcast has a higher priority than the task sending the message (or interrupted task, if the message is sent by an ISR), C/OS-III will run the highest-priority task that is waiting. Notice that not all tasks must specify a timeout; some tasks may want to wait forever.

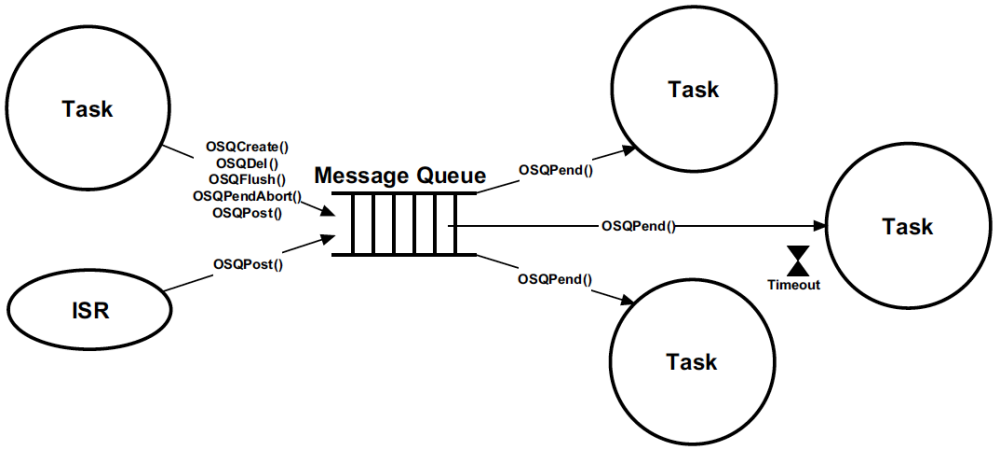


Figure - Multiple tasks waiting on a message queue

Task Message Queue

It is fairly rare to find applications where multiple tasks wait on a single message queue. Because of this, a message queue is built into each task and the user can send messages directly to a task without going through an external message queue object. This feature not only simplifies the code but, is also more efficient than using a separate message queue object. The message queue that is built into each task is shown in the figure below.

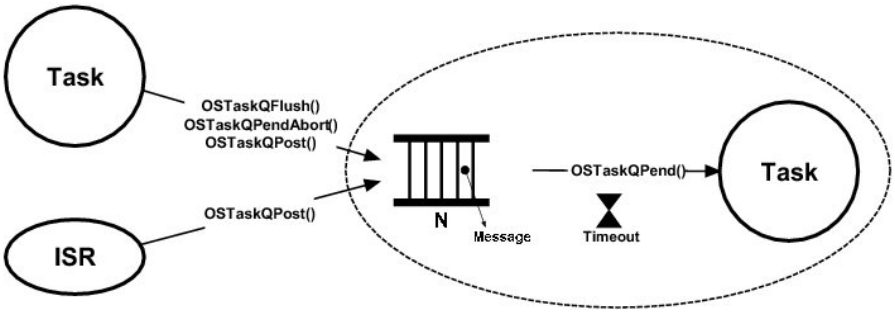


Figure - Task message queue

Task message queue services in µC/OS-III start with the `OSTaskQ???()` prefix, and services available to the application programmer are described in µC/OS-III API Reference. Setting `OS_CFG_TASK_Q_EN` to `DEF_ENABLED` in `os_cfg.h` enables task message queue services. The code for task message queue management is found in `os_task.c`.

You use this feature if the code knows which task to send the message(s) to. For example, if receiving an interrupt from an Ethernet controller, you can send the address of the received packet to the task that will be responsible for processing the received packet.

Copyright 2015 Micrium Inc.

Bilateral Rendez-vous

Two tasks can synchronize their activities by using two message queues, as shown in the figure below. This is called a *bilateral rendez-vous* and works the same as with semaphores except that both tasks may send messages to each other. A bilateral rendez-vous cannot be performed between a task and an ISR since an ISR cannot wait on a message queue.

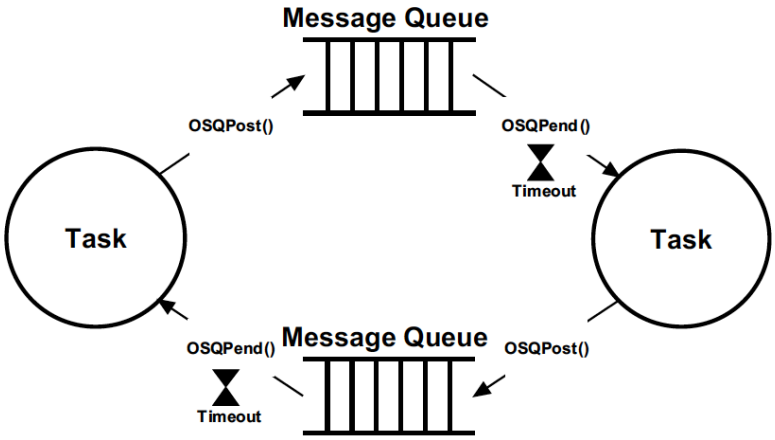


Figure - Bilateral Rendez-vous

In a bilateral rendez-vous, each message queue holds a maximum of one message. Both message queues are initially created empty. When the task on the left reaches the rendez-vous point, it sends a message to the top message queue and waits for a message to arrive on the bottom message queue. Similarly, when the task on the right reaches its rendez-vous point, it sends a message to the message queue on the bottom and waits for a message to arrive on the top message queue.

The figure below shows how to use task-message queues to perform a bilateral rendez-vous.

Copyright 2015 Micrium Inc.

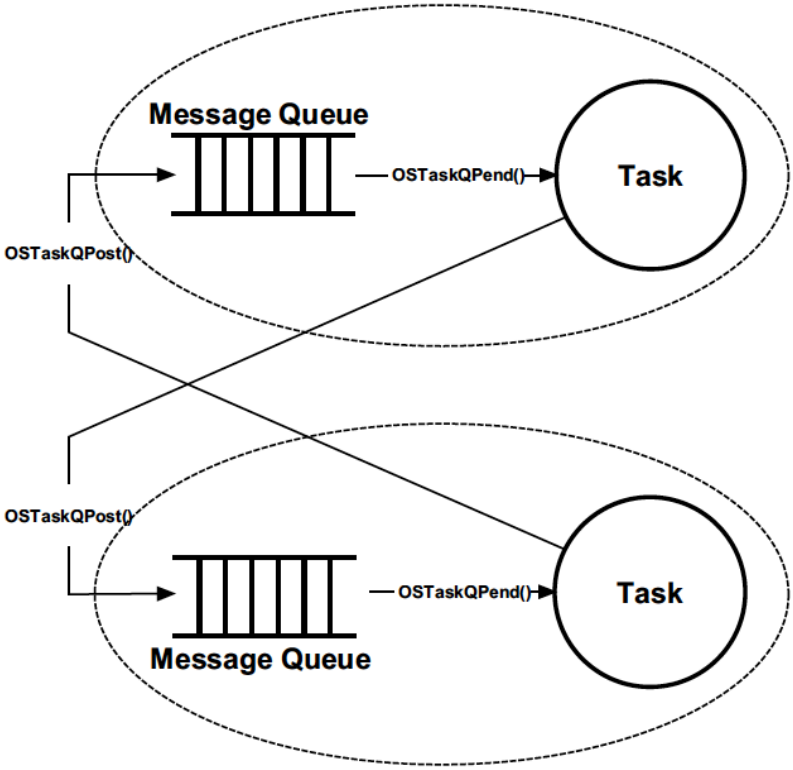


Figure - Figure Bilateral Rendez-vous with task message queues

Flow Control

Task-to-task communication often involves data transfer from one task to another. One task produces data while the other *consumes* it. However, data processing takes time and consumers might not consume data as fast as it is produced. In other words, it is possible for the producer to overflow the message queue if a higher-priority task preempts the consumer. One way to solve this problem is to add flow control in the process as shown in the figure below.

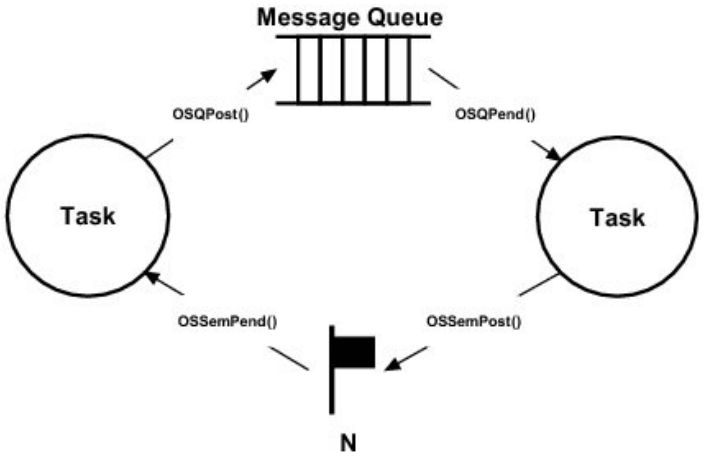


Figure - Producer and consumer tasks with flow control

Here, a counting semaphore is used, initialized with the number of allowable messages that can be sent by the consumer. If the consumer cannot queue more than 10 messages, the counting semaphore contains a count of 10.

As shown in the pseudo code of the listing below, the producer must wait on the semaphore before it is allowed to send a message. The consumer waits for messages and, when processed, signals the semaphore.

```
Producer Task:
    Pend on Semaphore;
    Send message to message queue;

Consumer Task:
    Wait for message from message queue;
    Signal the semaphore;
```

Listing - Producer and consumer flow control

Combining the task message queue and task semaphores (see Synchronization), it is easy to

implement flow control as shown in the figure below. In this case, however, `OSTaskSemSet()` must be called immediately after creating the task to set the value of the task semaphore to the same value as the maximum number of allowable messages in the task message queue.

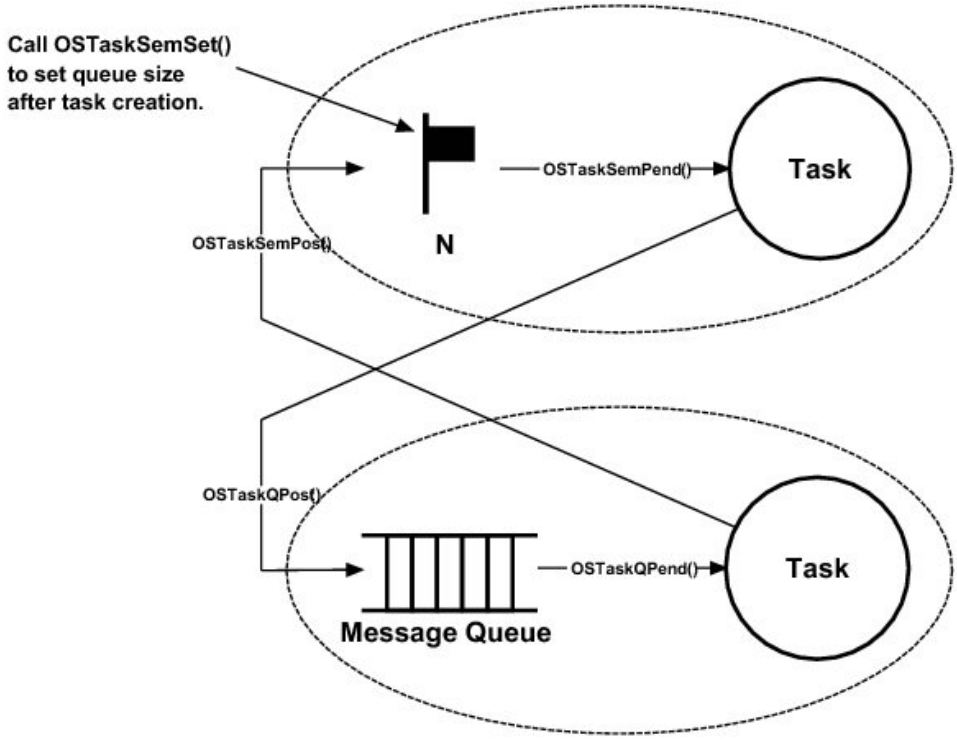


Figure - Flow control with task semaphore and task message queue

Keeping the Data in Scope

The messages sent typically point to data structures, variables, arrays, tables, etc. However, it is important to realize that the data must remain static until the receiver of the data completes its processing of the data. Once sent, the sender must not touch the sent data. This seems obvious, however it is easy to forget.

One possibility is to use the fixed-size memory partition manager provided with µC/OS-III (see Memory Management API Changes) to dynamically allocate and free memory blocks used to pass the data. The figure below shows an example. For sake of illustration, assume that a device is sending data bytes to the UART in packets using some protocol. In this case, the first byte of a packet is unique and the end-of-packet byte is also unique.

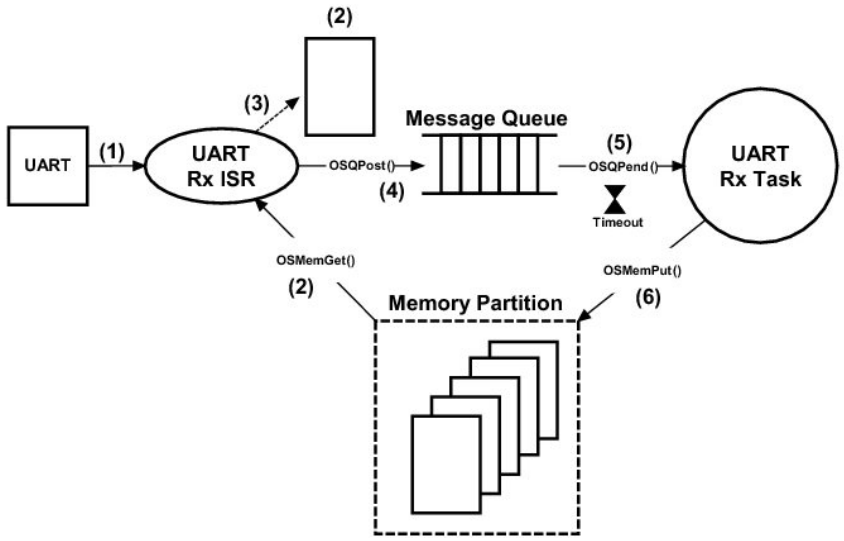


Figure - Using memory partitions for message contents

(1) Here, a UART generates an interrupt when characters are received.

(2) The pseudo-code in the listing below shows what the UART ISR code might look like. There are a lot of details omitted for sake of simplicity. The ISR reads the byte received from the UART and sees if it corresponds to a start of packet. If it is, a buffer is obtained from the memory partition.

(3) The received byte is then placed in the buffer.

Copyright 2015 Micrium Inc.

(4) If the data received is an end-of-packet byte, you would simply post the address of the buffer to the message queue so that the task can process the received packet.

(5) If the message sent makes the UART task the highest priority task, µC/OS-III will switch to that task at the end of the ISR instead of returning to the interrupted task. The task retrieves the packet from the message queue. Note that the `OSQPend()` call also returns the number of bytes in the packet and a time stamp indicating when the message was sent.

(6) When the task is finished processing the packet, the buffer is returned to the memory partition it came from by calling `OSMemPut()`.

```
void  UART_ISR (void)
          {
              OS_ERR  err;


              RxData = Read byte from UART;
              if (RxData == Start of Packet) {               /* See if we need a new buffer      */
                  RxDataPtr = OSMemGet(&UART_MemPool,        /* Yes                              */
                                       &err);
                 *RxDataPtr++ = RxData;
                  RxDataCtr   = 1;
              } if (RxData == End of Packet byte) {          /* See if we got a full packet      */
                 *RxDataPtr++ = RxData;
                  RxDataCtr++;
                  OSQPost((OS_Q      *)&UART_Q,              /* Yes, post to task for processing */
                          (void      *)RxDataPtr,
                          (OS_MSG_SIZE)RxDataCtr,
                          (OS_OPT     )OS_OPT_POST_FIFO,
                          (OS_ERR    *)&err);
                  RxDataPtr = NULL;                         /* Don't point to sent buffer       */
                  RxDataCtr = 0;
              } else; {
                 *RxDataPtr++ = RxData;                      /* Save the byte received           */
                  RxDataCtr++;
             }
          }
```

Listing - UART ISR Pseudo-code

Using Message Queues

The table below shows a summary of message-queue services available from µC/OS-III. Refer to µC-OS-III API Reference for a full description on their use.

Function Name	Operation
`OSQCreate()`	Create a message queue.
`OSQDel()`	Delete a message queue.
`OSQFlush()`	Empty the message queue.
`OSQPend()`	Wait for a message.
`OSQPendAbort()`	Abort waiting for a message.
`OSQPost()`	Send a message through a message queue.

Table - Message queue API summary

The table below is a summary of task message queue services available from µC/OS-III. Refer to µC-OS-III API Reference , for a full description of their use.

Function Name	Operation
`OSTaskQPend()`	Wait for a message.
`OSTaskQPendAbort()`	Abort the wait for a message.
`OSTaskQPost()`	Send a message to a task.
`OSTaskQFlush()`	Empty the message queue.

Table - Task message queue API summary

The figure below shows an example of using a message queue when determining the speed of a rotating wheel.

Copyright 2015 Micrium Inc.

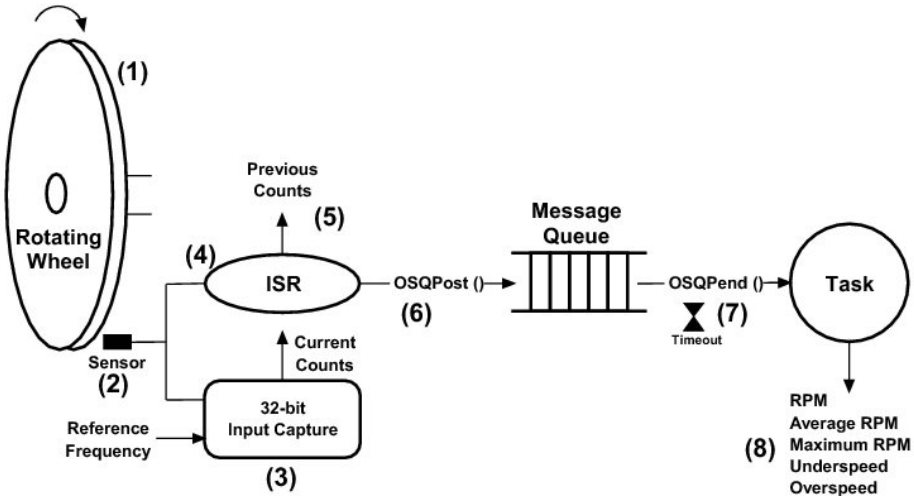


Figure - Measuring RPM

(1) The goal is to measure the RPM of a rotating wheel.

(2) A sensor is used to detect the passage of a hole in the wheel. In fact, to receive additional resolution, the wheel could contain multiple holes that are equally spaced.

(3) A 32-bit input capture register is used to capture the value of a free-running counter when the hole is detected.

(4) An interrupt is generated when the hole is detected. The ISR reads the current count of the input capture register and subtracts the value of the previous capture to determine the time it took for one rotation (assuming only a single hole).

```
Delta Counts = Current Counts – Previous Counts;
Previous Counts = Current Counts;
```

(5- 6) The delta counts are sent to a message queue. Since a message is actually a pointer, if the pointer is 32-bits wide on the processor in use, you can simply cast the 32-bit delta counts to a pointer and send this through the message queue. A safer and more portable approach is to dynamically allocate storage to hold the delta counts using a memory block from µC/OS-III's memory management services (see Memory Management API Changes) and send the address of the allocated memory block. The counts read are then saved in 'Previous Counts' to be used on the next interrupt.

Copyright 2015 Micrium Inc.

(7) When the message is sent, the RPM measurement task wakes up and computes the RPM as follows:

```
RPM = 60 * Reference Frequency / Delta Counts;
```

The user may specify a timeout on the pend call and the task will wake up if a message is not sent within the timeout period. This allows the user to easily detect that the wheel is not rotating and therefore, the RPM is 0.

(8) Along with computing RPM, the task can also compute average RPM, maximum RPM, and whether the speed is above or below thresholds, etc.

A few interesting things are worth noting about the above example. First, the ISR is very short; it reads the input capture and post the delta counts to the task so it can computer the time-consuming math. Second, with the timeout on the pend, it is easy to detect that the wheel is stopped. Finally, the task can perform additional calculations and can further detect such errors as the wheel spinning too fast or too slow. In fact, the task can notify other tasks about these errors, if needed.

The listing below shows how to implement the RPM measurement example using µC/OS-III's message queue services. Some of the code is pseudo-code, while the calls to µC/OS-III services are actual calls with their appropriate arguments.

Copyright 2015 Micrium Inc.

```
OS_Q        RPM_Q;                                                  (1)
CPU_INT32U  DeltaCounts;
CPU_INT32U  CurrentCounts;
CPU_INT32U  PreviousCounts;


void main (void)
{
    OS_ERR  err ;
    :
    OSInit(&err) ;                                                  (2)
    :
    OSQCreate((OS_Q     *)&RPM_Q,
              (CPU_CHAR *)”My Queue”,
              (OS_MSG_QTY)10,
              (OS_ERR   *)&err);
    :
    OSStart(&err);
}


void RPM_ISR (void)                                                 (3)
{
    OS_ERR  err;


    Clear the interrupt from the sensor;
    CurrentCounts  = Read the input capture;
    DeltaCounts    = CurrentCounts – PreviousCounts;
    PreviousCounts = CurrentCounts;
    OSQPost((OS_Q      *)&RPM_Q,                                    (4)
            (void      *)DeltaCounts,
            (OS_MSG_SIZE)sizeof(void *),
            (OS_OPT     )OS_OPT_POST_FIFO,
            (OS_ERR    *)&err);
}

void RPM_Task (void *p_arg)
{
    CPU_INT32U   delta;
    OS_ERR       err;
    OS_MSG_SIZE  size;
    CPU_TS       ts;


    DeltaCounts    = 0;
    PreviousCounts = 0;
    CurrentCounts  = 0;
    while (DEF_ON) {
        delta = (CPU_INT32U)OSQPend((OS_Q        *)&RPM_Q,          (5)
                                    (OS_TICK      )OS_CFG_TICK_RATE_HZ * 10,
                                    (OS_OPT       )OS_OPT_PEND_BLOCKING,
                                    (OS_MSG_SIZE *)&size,
                                    (CPU_TS      *)&ts,
                                    (OS_ERR      *)&err);
        if (err == OS_ERR_TIMEOUT) {                                (6)
            RPM = 0;
        } else {
            if (delta > 0u) {
                RPM = 60 * Reference Frequency / delta;             (7)
            }
        }
        Compute average RPM;                                        (8)
        Detect maximum RPM;
        Check for overspeed;
        Check for underspeed;
        :
```

```
        :
    }
}
```

Listing - Pseudo-code of RPM measurement

(1) Variables are declared. Notice that it is necessary to allocate storage for the message queue itself.

(2) You need to call `OSInit()` and create the message queue before it is used. The best place to do this is in startup code.

(3) The RPM ISR clears the sensor interrupt and reads the value of the 32-bit input capture. Note that it is possible to read RPM if there is only a 16-bit input capture. The problem with a 16-bit input capture is that it is easy for it to overflow, especially at low RPMs.

The RPM ISR also computes delta counts directly in the ISR. It is just as easy to post the current counts and let the task compute the delta. However, the subtraction is a fast operation and does not significantly increase ISR processing time.

(4) The code then sends the delta counts to the RPM task, which is responsible for computing the RPM and perform additional computations. Note that the message gets lost if the queue is full when the user attempts to post. This happens if data is generated faster than it is processed. Unfortunately, it is not possible to implement flow control in the example because we are dealing with an ISR.

(5) The RPM task starts by waiting for a message from the RPM ISR by pending on the message queue. The third argument specifies the timeout. In this case, ten seconds worth of timeout is specified. However, the value chosen depends on the requirements of an application.

Also notice that the `ts` variable contains the timestamp of when the post was completed. You can determine the time it took for the task to respond to the message received by calling `OS_TS_GET()`, and subtract the value of `ts`:

```
response_time = OS_TS_GET() – ts;
```

(6) If a timeout occurs, you can assume the wheel is no longer spinning.

Copyright 2015 Micrium Inc.

(7) The RPM is computed from the delta counts received, and from the reference frequency of the free-running counter.

(8) Additional computations are performed as needed. In fact, messages can be sent to different tasks in case error conditions are detected. The messages would be processed by the other tasks. For example, if the wheel spins too fast, another task can initiate a shutdown on the device that is controlling the wheel speed.

In the lising below , `OSQPost()` and `OSQPend()` are replaced with `OSTaskQPost()` and `OSTaskQPend()` for the RPM measurement example. Notice that the code is slightly simpler to use and does not require creating a separate message queue object. However, when creating the RPM task, it is important to specify the size of the message queue used by the task and compile the application code with `OS_CFG_TASK_Q_EN` set to `DEF_ENABLED`. The differences between using message queues and the task's message queue will be explained.

Copyright 2015 Micrium Inc.

```
OS_TCB RPM_TCB; (1)
OS_STK      RPM_Stk[1000];
CPU_INT32U  DeltaCounts ;
CPU_INT32U  CurrentCounts ;
CPU_INT32U  PreviousCounts ;


void main (void)
{
    OS_ERR  err ;
    :
    OSInit(&err) ;
    :
    void  OSTaskCreate ((OS_TCB       *)&RPM_TCB,                   (2)
                        (CPU_CHAR     *)”RPM Task”,
                        (OS_TASK_PTR   )RPM_Task,
                        (void         *)0,
                        (OS_PRIO       )10,
                        (CPU_STK      *)&RPM_Stk[0],
                        (CPU_STK_SIZE  )100,
                        (CPU_STK_SIZE  )1000,
                        (OS_MSG_QTY    )10,
                        (OS_TICK       )0,
                        (void         *)0,
                        (OS_OPT        )(OS_OPT_TASK_STK_CHK + OS_OPT_TASK_STK_CLR),
                        (OS_ERR       *)&err);
    :
    OSStart(&err);
}

void RPM_ISR (void)
{
    OS_ERR  err;

    Clear the interrupting from the sensor;
    CurrentCounts  = Read the input capture;
    DeltaCounts    = CurrentCounts – PreviousCounts;
    PreviousCounts = CurrentCounts;
    OSTaskQPost((OS_TCB    *)&RPM_TCB,                                  (3)
                (void      *)DeltaCounts,
                (OS_MSG_SIZE)sizeof(DeltaCounts),
                (OS_OPT     )OS_OPT_POST_FIFO,
                (OS_ERR    *)&err);
}

void RPM_Task (void *p_arg)
{
    CPU_INT32U   delta;
    OS_ERR       err;
    OS_MSG_SIZE  size;
    CPU_TS       ts;


    DeltaCounts    = 0;
    PreviousCounts = 0;
    CurrentCounts  = 0;
    while (DEF_ON) {
        delta = (CPU_INT32U)OSTaskQPend((OS_TICK       )OS_CFG_TICK_RATE * 10,   (4)
                                        (OS_OPT       )OS_OPT_PEND_BLOCKING,
                                        (OS_MSG_SIZE *)&size,
                                        (CPU_TS      *)&ts,
                                        (OS_ERR      *)&err);
        if (err == OS_ERR_TIMEOUT) {
            RPM = 0;
        } else {
            if (delta > 0u) {
                RPM = 60 * ReferenceFrequency / delta;
```

```
            }
        }
        Compute average RPM;
        Detect maximum RPM;
        Check for overspeed;
        Check for underspeed;
        :
        :
    }
}
```

Listing - Pseudo-code of RPM measurement

(1) Instead of declaring a message queue, it is important to know the `OS_TCB` of the task that will be receiving messages.

(2) The RPM task is created and a queue size of 10 entries is specified. Of course, hard-coded values should not be specified in a real application, but instead, you should use `#defines`. Fixed numbers are used here for sake of illustration.

(3) Instead of posting to a message queue, the ISR posts the message directly to the task, specifying the address of the `OS_TCB` of the task. This is known since the `OS_TCB` is allocated when creating the task.

(4) The RPM task starts by waiting for a message from the RPM ISR by calling `OSTaskQPend()`. This is an inherent call so it is not necessary to specify the address of the `OS_TCB` to pend on as the current task is assumed. The second argument specifies the timeout. Here, ten seconds worth of timeout is specified, which corresponds to 6 RPM.

Copyright 2015 Micrium Inc.

Clients and Servers

Another interesting use of message queues is shown in the figure below. Here, a task (the server) is used to monitor error conditions that are sent to it by other tasks or ISRs (clients). For example, a client detects whether the RPM of the rotating wheel has been exceeded, another client detects whether an over-temperature exists, and yet another client detects that a user pressed a shutdown button. When the clients detect error conditions, they send a message through the message queue. The message sent indicates the error detected, which threshold was exceeded, the error code that is associated with error conditions, or even suggests the address of a function that will handle the error, and more.

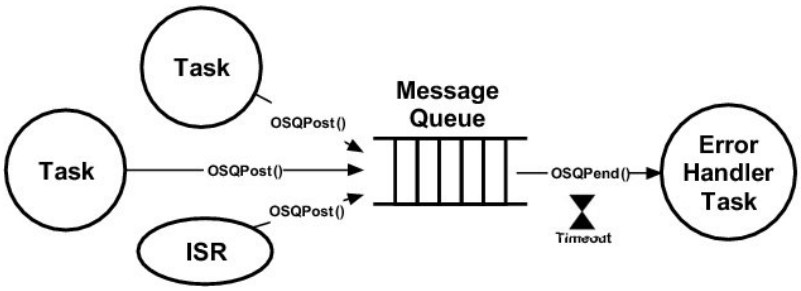


Figure - Clients and Servers

Message Queues Internals

As previously described, a message consists of a pointer to actual data, a variable indicating the size of the data being pointed to and a timestamp indicating when the message was actually sent. When sent, a message is placed in a data structure of type `OS_MSG`, shown in the figure below.

The sender and receiver are unaware of this data structure since everything is hidden through the APIs provided by µC/OS-III.

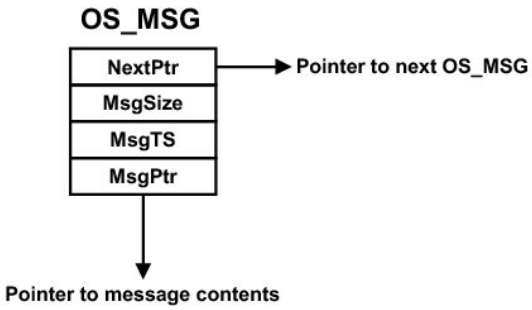


Figure - OS_MSG structure

µC/OS-III maintains a pool of free `OS_MSGs`. The total number of available messages in the pool is determined by the value of `OS_CFG_MSG_POOL_SIZE` found in `os_cfg_app.h`. When µC/OS-III is initialized, `OS_MSGs` are linked in a single linked list as shown in the figure below. Notice that the free list is maintained by a data structure of type `OS_MSG_POOL`, which contains four (4) fields: `.NextPtr`, which points to the free list; `.NbrFree`, which contains the number of free `OS_MSGs` in the pool, `.NbrUsed`, which contains the number of `OS_MSGs` allocated to the application and, `.NbrUsedMax` which detects the maximum number of messages allocated to the application.

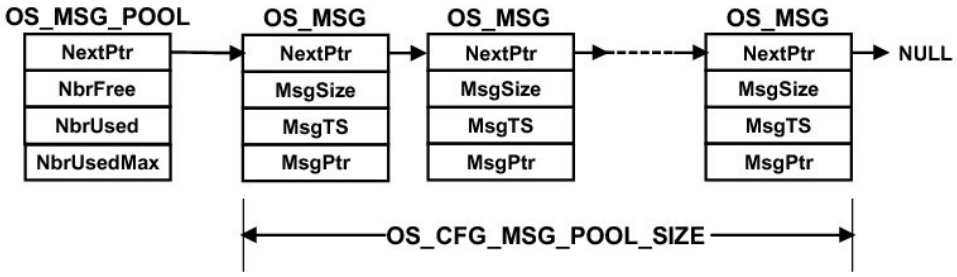


Figure - Pool of free OS_MSGs

Messages are queued using a data structure of type `OS_MSG_Q`, as shown in the figure below.

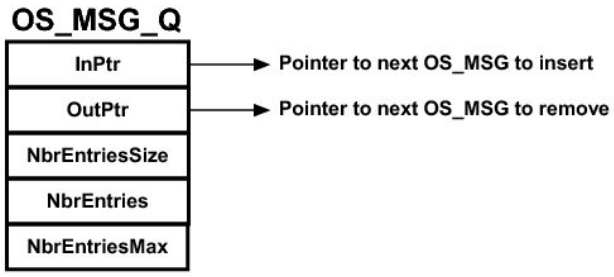


Figure - OS_MSG_Q structure

`.InPtr`

This field contains a pointer to where the next `OS_MSG` will be inserted in the queue. In fact, the `OS_MSG` will be inserted “after” the `OS_MSG` pointed to.

`.OutPtr`

This field contains a pointer to where the next `OS_MSG` will be extracted.

`.NbrEntriesSize`

This field contains the maximum number of `OS_MSGs` that the queue will hold. If an application attempts to send a message and the `.NbrEntries` matches this value, the queue is considered to be full and the `OS_MSG` will not be inserted.

`.NbrEntries`

This field contains the current number of `OS_MSGs` in the queue.

`.NbrEntriesMax`

This field contains the highest number of `OS_MSGs` existing in the queue at any given time.

A number of internal functions are used by µC/OS-III to manipulate the free list and messages. Specifically, `OS_MsgQPut()` inserts an `OS_MSG` in an `OS_MSG_Q`, `OS_MsgQGet()` extracts an `OS_MSG` from an `OS_MSG_Q`, and `OS_MsgQFreeAll()` returns all `OS_MSGs` in an `OS_MSG_Q` to the pool of free `OS_MSGs`. There are other `OS_MsgQ??()` functions in `os_msg.c` that are used during initialization.

The figure below shows an example of an `OS_MSG_Q` when four `OS_MSGs` are inserted.

Copyright 2015 Micrium Inc.

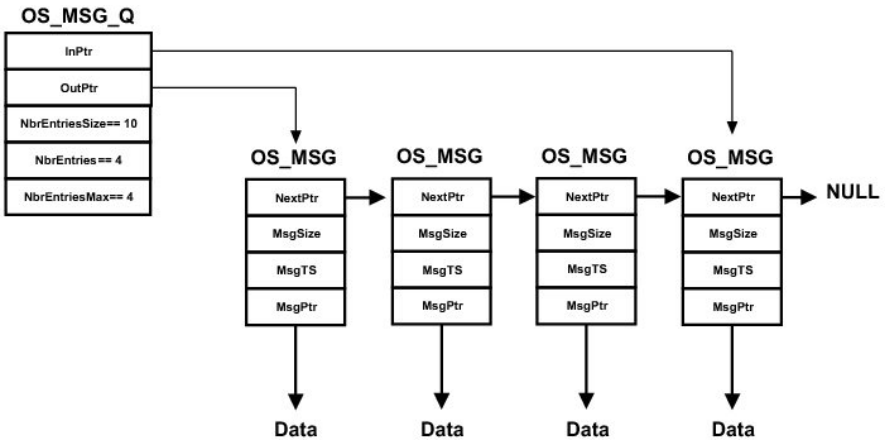


Figure - OS_MSG_Q with four OS_MSGs

`OS_MSG_Q`s are used inside two additional data structures: `OS_Q` and `OS_TCB`. Recall that an `OS_Q` is declared when creating a message queue object. An `OS_TCB` is a task control block and, as previously mentioned, each `OS_TCB` can have its own message queue when the configuration constant `OS_CFG_TASK_Q_EN` is set to `DEF_ENABLED` in `os_cfg.h`. The figure below shows the contents of an `OS_Q` and partial contents of an `OS_TCB` containing an `OS_MSG_Q`. The `OS_MSG_Q` data structure is shown as an “exploded view” to emphasize the structure within the structure.

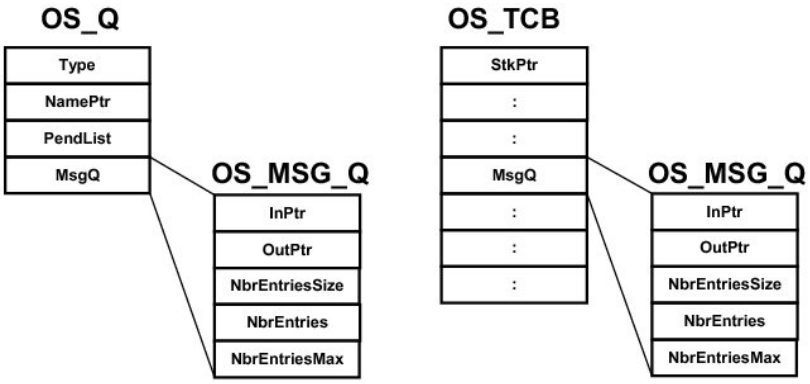


Figure - OS_Q and OS_TCB each contain an OS_MSG_Q

Pending on Multiple Objects

In Pend Lists we saw how multiple tasks can pend (or wait) on a single kernel object such as a semaphore, mutual exclusion semaphore, event flag group, or message queue. In this chapter, we will see how tasks can pend on multiple objects. However, µC/OS-III only allows for pend on multiple semaphores and/or message queues. In other words, it is not possible to pend on multiple event flag groups or mutual exclusion semaphores.

As shown in the figure below, a task can pend on any number of semaphores or message queues at the same time. The first semaphore or message queue posted will make the task ready-to-run and compete for CPU time with other tasks in the ready list. As shown, a task pends on multiple objects by calling `OSPendMulti()` and specifies an optional timeout value. The timeout applies to all of the objects. If none of the objects are posted within the specified timeout, the task resumes with an error code indicating that the pend timed out.

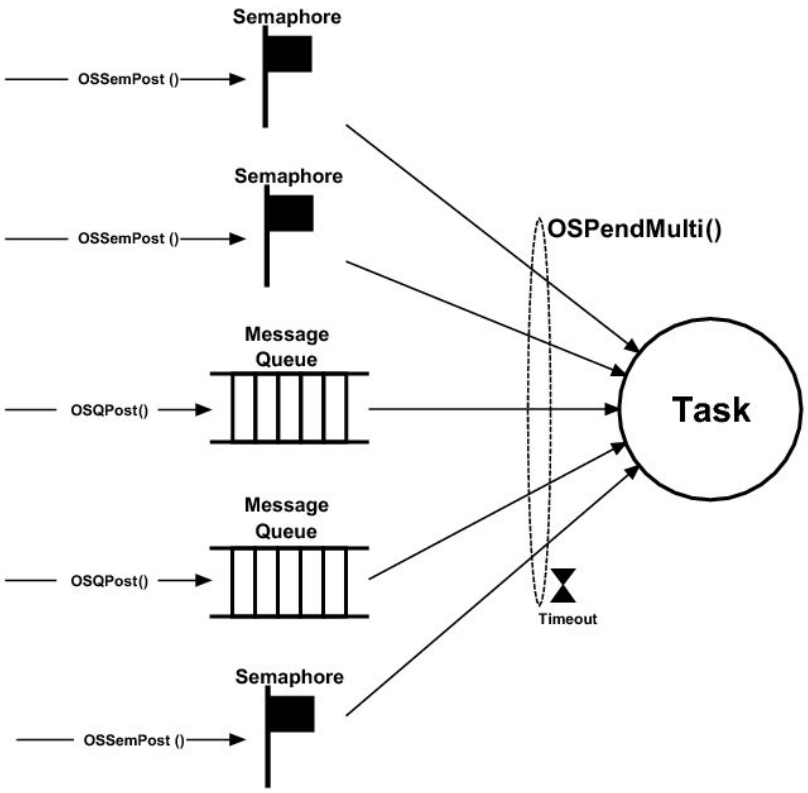


Figure - Task pending on multiple objects

The listing below shows the function prototype of `OSPendMulti()` and the figure that follows shows an array of `OS_PEND_DATA` elements.

```
OS_OBJ_QTY  OSPendMulti (OS_PEND_DATA  *p_pend_data_tbl,                  (1)
                        OS_OBJ_QTY     tbl_size,                          (2)
                        OS_TICK        timeout,                           (3)
                        OS_OPT         opt,                               (4)
                        OS_ERR        *p_err);                            (5)
```

Listing - OSPendMulti() prototype

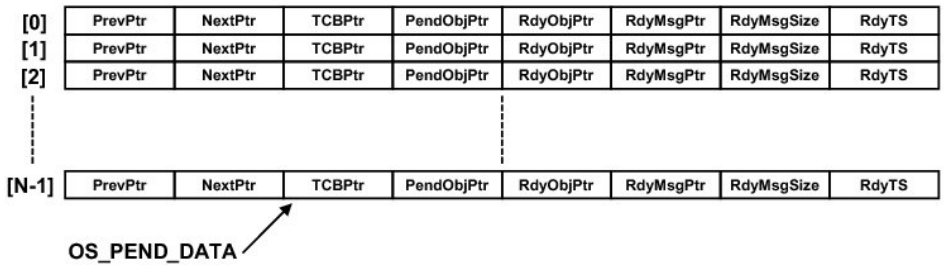


Figure - Array of OS_PEND_DATA

(1) `OSPendMulti()` is passed an array of `OS_PEND_DATA` elements. The caller must instantiate an array of `OS_PEND_DATA`. The size of the array depends on the total number of kernel objects that the task wants to pend on. For example, if the task wants to pend on three semaphores and two message queues then the array contains five `OS_PEND_DATA` elements as shown below:

```
OS_PEND_DATA my_pend_multi_tbl[5];
```

The calling task needs to initialize the `.PendObjPtr` of each element of the array to point to each of the objects to be pended on. For example:

```
OS_SEM  MySem1;
OS_SEM  MySem2;
OS_SEM  MySem3;
OS_Q    MyQ1;
OS_Q    MyQ2;


void  MyTask (void)
{
    OS_ERR        err;
    OS_PEND_DATA  my_pend_multi_tbl[5];
    :
    while (DEF_ON) {
        :
        my_pend_multi_tbl[0].PendObjPtr = (OS_PEND_OBJ)&MySem1;
        my_pend_multi_tbl[1].PendObjPtr = (OS_PEND_OBJ)&MySem2;
```

```
        my_pend_multi_tbl[2].PendObjPtr = (OS_PEND_OBJ)&MySem3;
        my_pend_multi_tbl[3].PendObjPtr = (OS_PEND_OBJ)&MyQ1;
        my_pend_multi_tbl[4].PendObjPtr = (OS_PEND_OBJ)&MyQ2;
        OSPendMulti((OS_PEND_DATA *)&my_pend_multi_tbl[0],
                    (OS_OBJ_QTY    )5,
                    (OS_TICK       )0,
                    (OS_OPT        )OS_OPT_PEND_BLOCKING,
                    (OS_ERR       *)&err);
        /* Check 'err" */
        :
    }
}
```

(2) This argument specifies the size of the `OS_PEND_DATA` table. In the above example, this is 5.

(3) You specify whether or not to timeout in case none of the objects are posted within a certain amount of time. A non-zero value indicates the number of ticks to timeout. Specifying zero indicates the task will wait forever for any of the objects to be posted.

(4) The "`opt`" argument specifies whether to wait for objects to be posted (you would set `opt` to `OS_OPT_PEND_BLOCKING`) or, not block if none of the objects have already been posted (you would set `opt` to `OS_OPT_PEND_NON_BLOCKING`).

(5) As with most µC/OS-III function calls, you specify the address of a variable that will receive an error code based on the outcome of the function call. See Appendix A, "µC/OS-III API Reference" for a list of possible error codes. As always, it is highly recommended to examine the error return code.

(6) Note that all objects are cast to `OS_PEND_OBJ` data types.

When called, `OSPendMulti()` first starts by validating that all of the objects specified in the `OS_PEND_DATA` table are either an `OS_SEM` or an `OS_Q` . If not, an error code is returned.

Next, `OSPendMulti()` goes through the `OS_PEND_DATA` table to see if any of the objects have already posted. If so, `OSPendMulti()` fills the following fields in the table: `.RdyObjPtr`, `.RdyMsgPtr`, `.RdyMsgSize` and `.RdyTS`.

`.RdyObjPtr`

is a pointer to the object if the object has been posted. For example, if the first object in the table is a semaphore and the semaphore has been posted to,

Copyright 2015 Micrium Inc.

my_pend_multi_tbl[0].RdyObjPtr is set to my_pend_multi_tbl[0].PendObjPtr.

.RdyMsgPtr

is a pointer to a message if the object in the table at this entry is a message queue and a message was received from the message queue.

.RdyMsgSize

is the size of the message received if the object in the table at this entry is a message queue and a message was received from the message queue.

.RdyTS

is the timestamp of when the object posted. This allows the user to know when a semaphore or message queue was posted.

If there are no objects posted, then OSPendMulti() places the current task in the wait list of all the objects that it is pending on. This is a complex and tedious process for OSPendMulti() since there can be other tasks in the pend list of some of these objects we are pending on.

To indicate how tricky things get, the figure below is an example of a task pending on two semaphores.

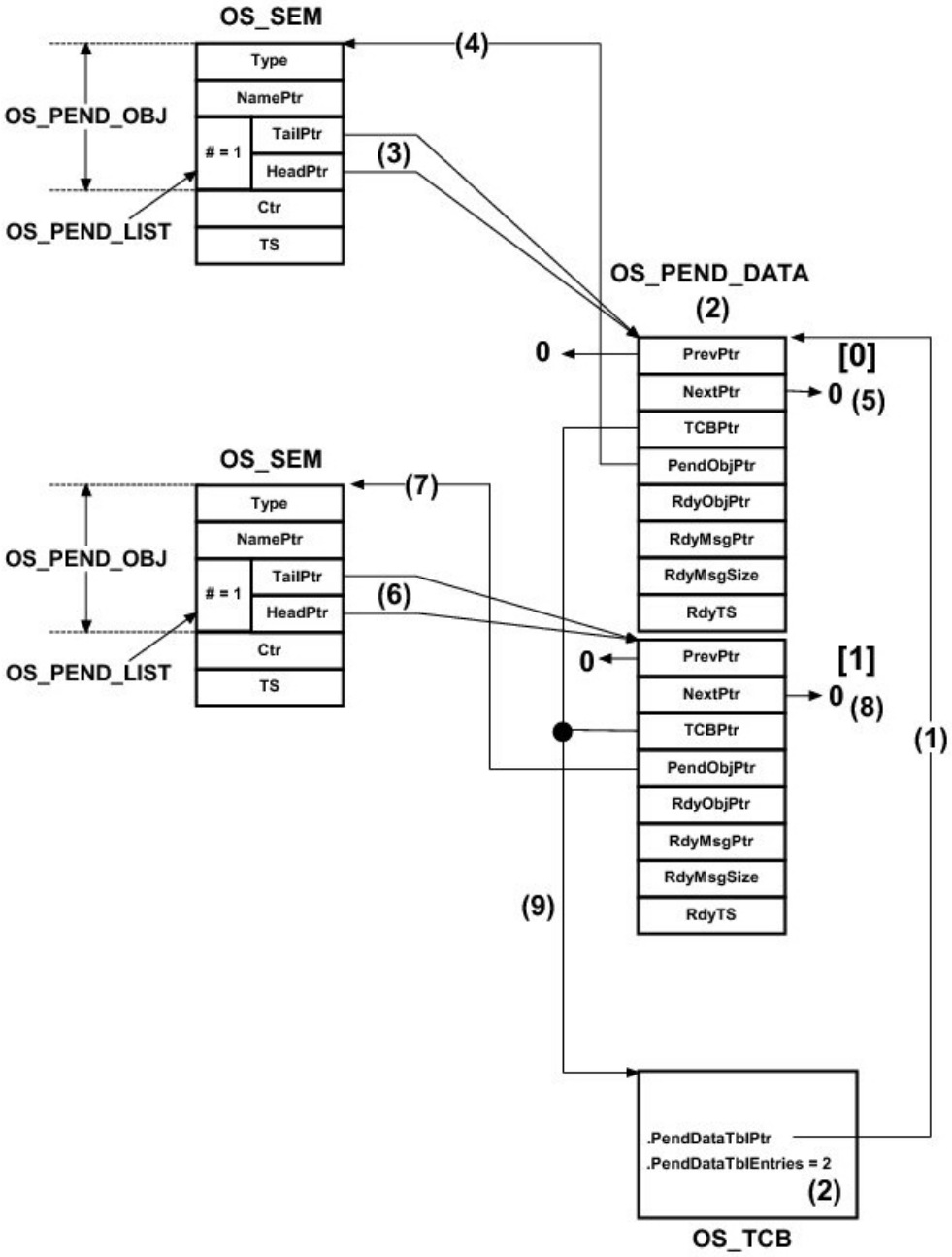


Figure - Task pending on two semaphores

(1) A pointer to the base address of the `OS_PEND_DATA` table is placed in the `OS_TCB` of the task placed in the pend list of the two semaphores.

(2) The number of entries in the `OS_PEND_DATA` table is also placed in the `OS_TCB`. Again, this

task is waiting on two semaphores and therefore there are two entries in the table.

(3) The first semaphore is linked to the first entry in the `OS_PEND_DATA` array.

(4) Entry [0] of the `OS_PEND_DATA` table is linked to the semaphore object specified by that entry's `.PendObjPtr`. This pointer was specified by the caller of `OSPendMulti()`.

(5) Since there is only one task in the pend list of the semaphore, the `.PrevPtr` and `.NextPtr` are pointing to `NULL`.

(6) The second semaphore points to the second entry in the `OS_PEND_DATA` table.

(7) The second entry in the `OS_PEND_DATA` array points to the second semaphore. This pointer was specified by the caller of `OSPendMulti()`.

(8) The second semaphore only has one entry in its pend list. Therefore the `.PrevPtr` and `.NextPtr` both point to `NULL`.

(9) `OSPendMulti()` links back each `OS_PEND_DATA` entry to the task that is waiting on the two semaphores.

The figure below is a more complex example where one task is pending on two semaphores while another task also pends on one of the two semaphores. The examples presented so far only show semaphores, but they could be combinations of semaphores and message queues.

Copyright 2015 Micrium Inc.

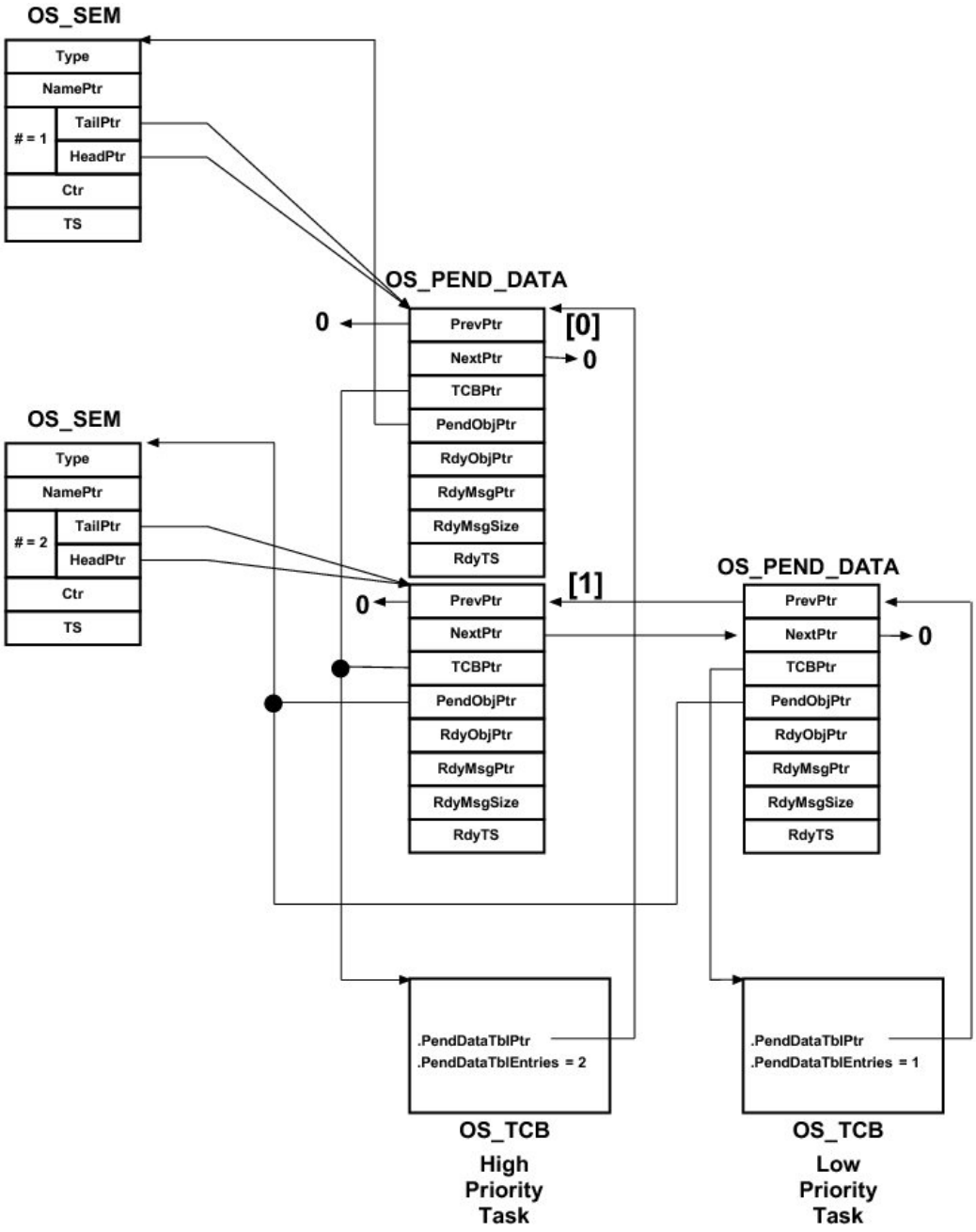


Figure - Tasks pending on semaphores

When either an ISR or a task signals or sends a message to one of the objects that the task is pending on, `OSPendMulti()` returns, indicating in the `OS_PEND_DATA` table which object was posted. This is done by only filling in “one” of the `.RdyObjPtr` entries, the one that corresponds to the object posted.

Only one of the entries in the `OS_PEND_DATA` table will have a `.RdyObjPtr` with a non-`NULL` value

while all the other entries have the `.RdyObjPtr` set to `NULL`. Going back to the case where a task waits on five semaphores and two message queues, if the first message queue is posted while the task is pending on all those objects, the `OS_PEND_DATA` table will be as shown in the figure below.

	.PrevPtr	.NextPtr	.TCBPtr	.PendObjPtr	.RdyObjPtr	.RdyMsgPtr	.RdyMsgSize	.RdyTS
[0]	0	0	TCBPtr	PendObjPtr	0	0	0	0
[1]	0	0	TCBPtr	PendObjPtr	0	0	0	0
[2]	0	0	TCBPtr	PendObjPtr	0	0	0	0
[3]	0	0	TCBPtr	&MyQ1	&MyQ1	Msg Ptr	Msg Size	Timestamp
[4]	0	0	TCBPtr	PendObjPtr	0	0	0	0

Figure - Message queue #1 posted before timeout expired

Copyright 2015 Micrium Inc.

Memory Management

An application can allocate and free dynamic memory using any ANSI C compiler's `malloc()` and `free()` functions, respectively. However, using `malloc()` and `free()` in an embedded real-time system may be dangerous. Eventually, it might not be possible to obtain a single contiguous memory area due to fragmentation. Fragmentation is the development of a large number of separate free areas (i.e., the total free memory is fragmented into small, non-contiguous pieces). Execution time of `malloc()` and `free()` is generally nondeterministic given the algorithms used to locate a contiguous block of free memory large enough to satisfy a `malloc()` request.

µC/OS-III provides an alternative to `malloc()` and `free()` by allowing an application to obtain fixed-sized memory blocks from a partition made from a contiguous memory area, as illustrated in the figure below. All memory blocks are the same size, and the partition contains an integral number of blocks. Allocation and deallocation of these memory blocks is performed in constant time and is deterministic. The partition itself is typically allocated statically (as an array), but can also be allocated by using `malloc()` as long as it is never freed.

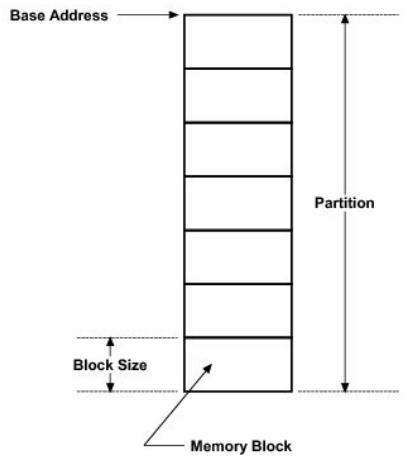


Figure - Memory Partition

As indicated in the figure below, more than one memory partition may exist in an application and each one may have a different number of memory blocks and be a different size. An application can obtain memory blocks of different sizes based upon requirements. However, a specific memory block must always be returned to the partition that it came from. This type of

memory management is not subject to fragmentation except that it is possible to run out of memory blocks. It is up to the application to decide how many partitions to have and how large each memory block should be within each partition.

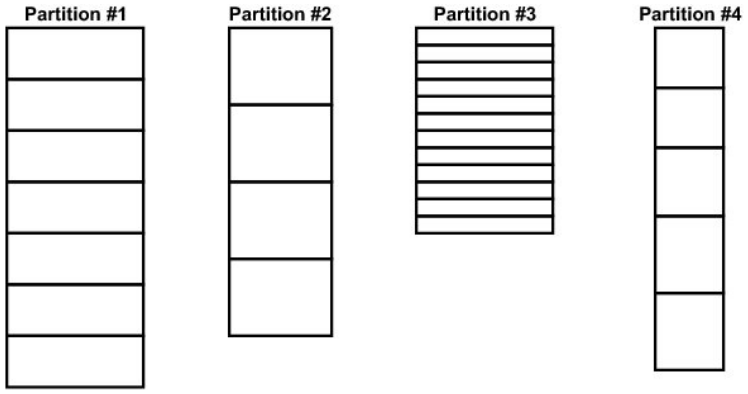


Figure - Multiple Memory Partitions

Creating a Memory Partition

Before using a memory partition, it must be created. This allows µC/OS-III to know something about the memory partition so that it can manage their allocation and deallocation. Once created, a memory partition is as shown in the figure below. Calling `OSMemCreate()` creates a memory partition.

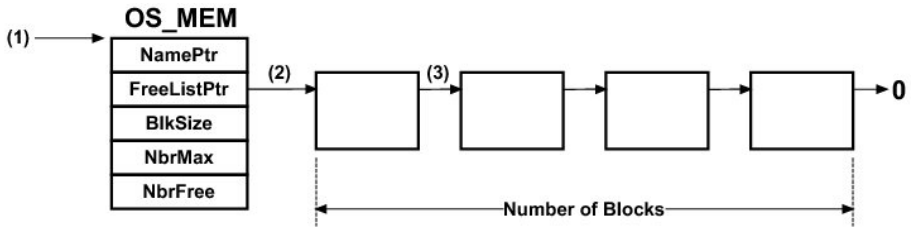


Figure - Created Memory Partition

(1) When creating a partition, the application code supplies the address of a memory partition control block (`OS_MEM`). Typically, this memory control block is allocated from static memory, however it can also be obtained from the heap by calling `malloc()`. The application code should however never deallocate it.

(2) `OSMemCreate()` organizes the continuous memory provided into a singly linked list and stores the pointer to the beginning of the list in the `OS_MEM` structure.

(3) Each memory block must be large enough to hold a pointer. Given the nature of the linked list, a block needs to be able to point to the next block.

The listing below indicates how to create a memory partition with µC/OS-III.

Copyright 2015 Micrium Inc.

```
OS_MEM       MyPartition;                                    (1)
          CPU_INT08U   MyPartitionStorage[12][100];                  (2)


          void  main (void)                                          (3)
          {
              OS_ERR  err;
              :
              :
              OSInit(&err);
              :
              OSMemCreate((OS_MEM    *)&MyPartition,                 (4)
                          (CPU_CHAR  *)”My Partition”,                (5)
                          (void      *)&MyPartitionStorage[0][0],     (6)
                          (OS_MEM_QTY ) 12,                           (7)
                          (OS_MEM_SIZE)100,                           (8)
                          (OS_ERR    *)&err);                         (9)
              /* Check ’err’ */
              :
              :
              OSStart(&err);
          }
```

Listing - Creating a memory partition

(1) An application needs to allocate storage for a memory partition control block (i.e. `OS_MEM` structure). This can be a static allocation as shown here or `malloc()` can be used in the code. However, the application code must not deallocate the memory control block.

(2) The application also needs to allocate storage for the memory that will be split into memory blocks. This can also be a static allocation or `malloc()` can be used. The same reasoning applies. Do not deallocate this storage since other tasks may rely on the existence of this storage.

(3) Memory partition must be created before allocating and deallocating blocks from the partition. One of the best places to create memory partitions is in `main()` prior to starting the multitasking process. Of course, an application can call a function from `main()` to do this instead of actually placing the code directly in `main()`.

(4) You pass the address of the memory partition control block to `OSMemCreate()`. You should never reference any of the internal members of the `OS_MEM` data structure. Instead, you should always use µC/OS-III’s API.

(5) You can assign a name to the memory partition. There is no limit to the length of the ASCII string as µC/OS-III saves a pointer to the ASCII string in the partition control block and not the actual characters.

Copyright 2015 Micrium Inc.

(6) You then need to pass the base address of the storage area reserved for the memory blocks.

(7) Here, you specify how many memory blocks are available from this memory partition. Hard coded numbers are used for the sake of the illustration but you should instead use `#define` constants.

(8) You need to specify the size of each memory block in the partition. Again, a hard coded value is used for illustration, which is not recommended in real code.

(9) As with most µC/OS-III services, `OSMemCreate()` returns an error code indicating the outcome of the service. The call is successful if "`err`" contains `OS_ERR_NONE`.

The listing below shows how to create a memory partition with µC/OS-III, but this time, using `malloc()` to allocate storage. Do not deallocate the memory control block or the storage for the partition.

```
OS_MEM      *MyPartitionPtr;                                        (1)


          void  main (void)
          {
              OS_ERR   err;
              void    *p_storage;
              :
              OSInit(&err);
              :
              MyPartitionPtr = (OS_MEM *)malloc(sizeof(OS_MEM));              (2)
              if (MyPartitionPtr != (OS_MEM *)0) {
                  p_storage = malloc(12 * 100);                               (3)
                  if (p_storage != (void *)0) {
                      OSMemCreate((OS_MEM    *)MyPartitionPtr,                (4)
                                  (CPU_CHAR  *)"My Partition",
                                  (void      *)p_storage,                     (5)
                                  (OS_MEM_QTY ) 12,                           (6)
                                  (OS_MEM_SIZE)100,                           (6)
                                  (OS_ERR    *)&err);
                      /* Check 'err" */
                  }
              }
              :
              OSStart(&err);
          }
```

Listing - Creating a memory partition

(1) Instead of allocating static storage for the memory partition control block, you can assign a pointer that receives the `OS_MEM` allocated using `malloc()`.

(2) The application allocates storage for the memory control block.

(3) We then allocate storage for the memory partition.

(4) A pointer is passed to the allocated memory control block to `OSMemCreate()`.

(5) The base address of the storage used for the partition is passed to `OSMemCreate()`.

(6) Finally, the number of blocks and the size of each block is passed so that µC/OS-III can create the linked list of 12 blocks of 100 bytes each. Again, hard coded numbers are used, but these would typically be replaced by `#defines`.

Copyright 2015 Micrium Inc.

Getting a Memory Block from a Partition

Application code can request a memory block from a partition by calling `OSMemGet()` as shown in the listing below. The code assumes that the partition was already created.

```
OS_MEM       MyPartition;                                                (1)
         CPU_INT08U  *MyDataBlkPtr;


         void  MyTask (void *p_arg)
         {
             OS_ERR  err;

             :
             while (DEF_ON) {
                 :
                 MyDataBlkPtr = (CPU_INT08U *)OSMemGet((OS_MEM    *)&MyPartition,   (2)
                                                       (OS_ERR    *)&err);
                 if (err == OS_ERR_NONE) {                                          (3)
                      /* You have a memory block from the partition */
                 }
                 :
                 :
             }
         }
```

Listing - Obtaining a memory block from a partition

(1) The memory partition control block must be accessible by all tasks or ISRs that will be using the partition.

(2) Simply call `OSMemGet()` to obtain a memory block from the desired partition. A pointer to the allocated memory block is returned. This is similar to `malloc()`, except that the memory block comes from a pool that is guaranteed to not fragment. Also, it’s assumed that your application knows the size of each block so it doesn’t overflow the block with data.

(3) It is important to examine the returned error code to ensure that there are free memory blocks and that the application can start putting content in the memory blocks.

Copyright 2015 Micrium Inc.

Returning a Memory Block to a Partition

The application code must return an allocated memory block back to the proper partition when finished. You do this by calling `OSMemPut()` as shown in the listing below. The code assumes that the partition was already created.

```
OS_MEM       MyPartition;                                          (1)
          CPU_INT08U  *MyDataBlkPtr;


          void  MyTask (void *p_arg)
          {
              OS_ERR  err;

              :
              while (DEF_ON) {
                  :
                  OSMemPut((OS_MEM  *)&MyPartition,                          (2)
                           (void    *)MyDataBlkPtr,                          (3)
                           (OS_ERR  *)&err);
                  if (err == OS_ERR_NONE) {                                  (4)
                       /* You properly returned the memory block to the partition */
                  }
                  :
                  :
              }
          }
```

Listing - Returning a memory block to a partition

(1) The memory partition control block must be accessible by all tasks or ISRs that will be using the partition.

(2) You simply call `OSMemPut()` to return the memory block back to the memory partition. Note that there is no check to see whether the proper memory block is being returned to the proper partition (assuming you have multiple different partitions). It is therefore important to be careful (as is necessary when designing embedded systems).

(3) You pass the pointer to the data area that is allocated so that it can be returned to the pool. Note that a "`void *`" is assumed.

(4) You would examine the returned error code to ensure that the call was successful.

Copyright 2015 Micrium Inc.

Using Memory Partitions

Memory management services are enabled at compile time by setting the configuration constant `OS_CFG_MEM_EN` to `DEF_ENABLED` in `os_cfg.h`.

There are a number of operations to perform on memory partitions as summarized in the table below.

Function Name	Operation
`OSMemCreate()`	Create a memory partition.
`OSMemGet()`	Obtain a memory block from a memory partition.
`OSMemPut()`	Return a memory block to a memory partition.

Table - Memory Partition API summary

`OSMemCreate()` can only be called from task-level code, but `OSMemGet()` and `OSMemPut()` can be called by Interrupt Service Routines (ISRs).

The listing show in Returning a Memory Block to a Partition shows an example of how to use the dynamic memory allocation feature of C/OS-III, as well as message-passing capabilities (see). In this example, the task on the left reads and checks the value of analog inputs (pressures, temperatures, and voltage) and sends a message to the second task if any of the analog inputs exceed a threshold. The message sent contains information about which channel had the error, an error code, an indication of the severity of the error, and other information.

Error handling in this example is centralized. Other tasks, or even ISRs, can post error messages to the error-handling task. The error-handling task could be responsible for displaying error messages on a monitor (a display), logging errors to a disk, or dispatching other tasks to take corrective action based on the error.

Copyright 2015 Micrium Inc.

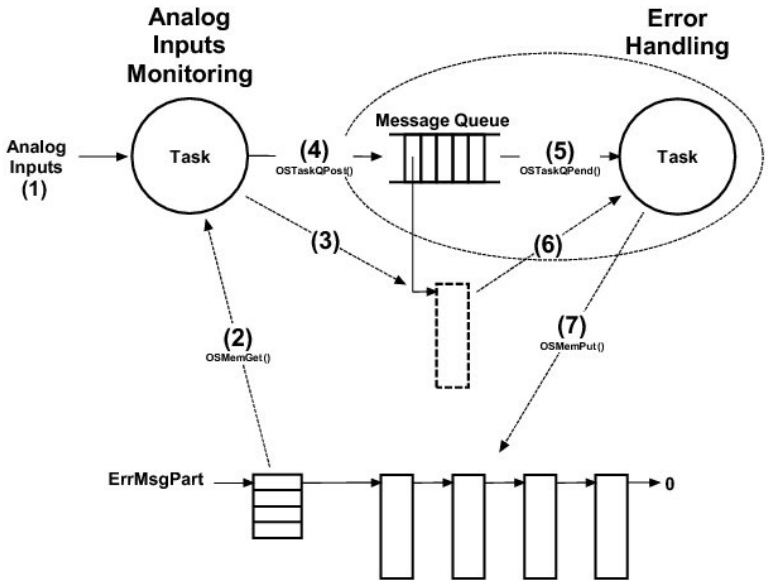


Figure - Using a Memory Partition – non blocking

(1) The analog inputs are read by the task. The task determines that one of the inputs is outside a valid range and an error message needs to be sent to the error handler.

(2) The task then obtains a memory block from a memory partition so that it can place information regarding the detected error.

(3) The task writes this information to the memory block. As mentioned above, the task places the analog channel that is at fault, an error code, an indication of the severity, possible solutions, and more. There is no need to store a timestamp in the message, as time stamping is a built-in feature of µC/OS-III so the receiving task will know when the message was posted.

(4) Once the message is complete, it is posted to the task that will handle such error messages. Of course the receiving task needs to know how the information is placed in the message. Once the message is sent, the analog input task is no longer allowed (by convention) to access the memory block since it sent it out to be processed.

(5) The error handler task (on the right) normally pends on the message queue. This task will not execute until a message is sent to it.

(6) When a message is received, the error handler task reads the contents of the message and

Copyright 2015 Micrium Inc.

performs necessary action(s). As indicated, once sent, the sender will not do anything else with the message.

(7) Once the error handler task is finished processing the message, it simply returns the memory block to the memory partition. The sender and receiver therefore need to know about the memory partition or, the sender can pass the address of the memory partition as part of the message and the error handler task will know where to return the memory block to.

Sometimes it is useful to have a task wait for a memory block in case a partition runs out of blocks. µC/OS-III does not support pending on partitions, but it is possible to support this requirement by adding a counting semaphore (see Resource Management) to guard the memory partition. The initial value of the counting semaphore would be set to the number of blocks in the partition. This is illustrated in the figure below.

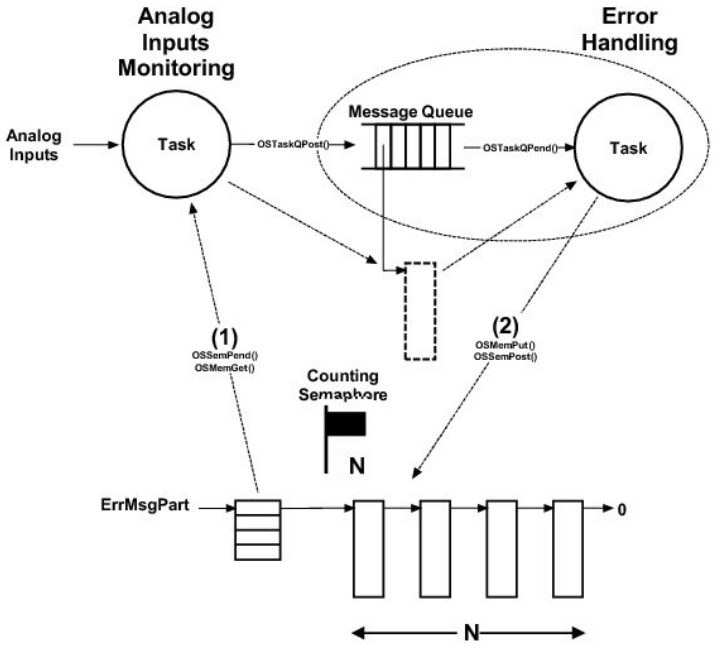


Figure - Using a Memory Partition - blocking

(1) To obtain a memory block, your code simply obtain the semaphore by calling `OSSemPend()` and then calls `OSMemGet()` to receive the memory block.

Copyright 2015 Micrium Inc.

(2) To release a block, you simply return the memory block by calling `OSMemPut()` and then signal the semaphore by calling `OSSemPost()`.

The above operations must be performed in order.

Note that the user may call `OSMemGet()` and `OSMemPut()` from an ISR since these functions do not block and in fact, execute very quickly. However, you cannot use blocking calls from ISRs.

Copyright 2015 Micrium Inc.

Porting uC-OS-III

This chapter describes how to adapt µC/OS-III to different processors. Adapting µC/OS-III to a microprocessor or a microcontroller is called porting. Most of µC/OS-III is written in C for portability. However, it is still necessary to write processor-speci?c code in C and assembly language. µC/OS-III manipulates processor registers, which can only be done using assembly language unless the C compiler supports inline assembly language extensions. Porting µC/OS-III to different processors is relatively easy as µC/OS-III was designed to be portable and, since µC/OS-III is similar to µC/OS-II, the user can start from a µC/OS-II port. In fact, this is the easiest way to do a µC/OS-III port.

If there is already a port for the processor to be used, it is not necessary to read this chapter unless, of course, there is an interest in knowing how µC/OS-III processor-speci?c code works.

µC/OS-III can run on a processor if it satisfies the following general requirements:

- The processor has an ANSI C compiler that generates reentrant code. In fact, the toolchain used must contain an assembler, C compiler and linker/locator. Finding such a toolchain is generally not an issue since there are a number of good toolchains available on the market.

- The processor supports interrupts and can provide an interrupt that occurs at regular intervals (typically between 10 and 1000 Hz). Most processors (especially MCUs) provide timer that can be used for this purpose. Some processors even have dedicated timers for use by an RTOS.

- Interrupts can be disabled and enabled. All current processors that we’ve worked with offer this. Ideally, the processor allows you to save the current state of the interrupt mask so that it can be restored.

- The processor supports a hardware stack that accommodates a fair amount of data (possibly many kilobytes).

- The processor has instructions to save and restore the stack pointer and other CPU registers, either on the stack or in memory.

- The processor has access to sufficient RAM for µC/OS-III’s variables and data structures as well as internal task stacks.

Copyright 2015 Micrium Inc.

- The compiler should support 32-bit data types. For some fast 32-bit processors, the compiler should also support 64-bit data types (typically “`long long`”).

The figure below shows the µC/OS-III architecture and its relationship with other software components and hardware. When using µC/OS-III in an application, the user is responsible for providing application software and the µC/OS-III configuration sections.

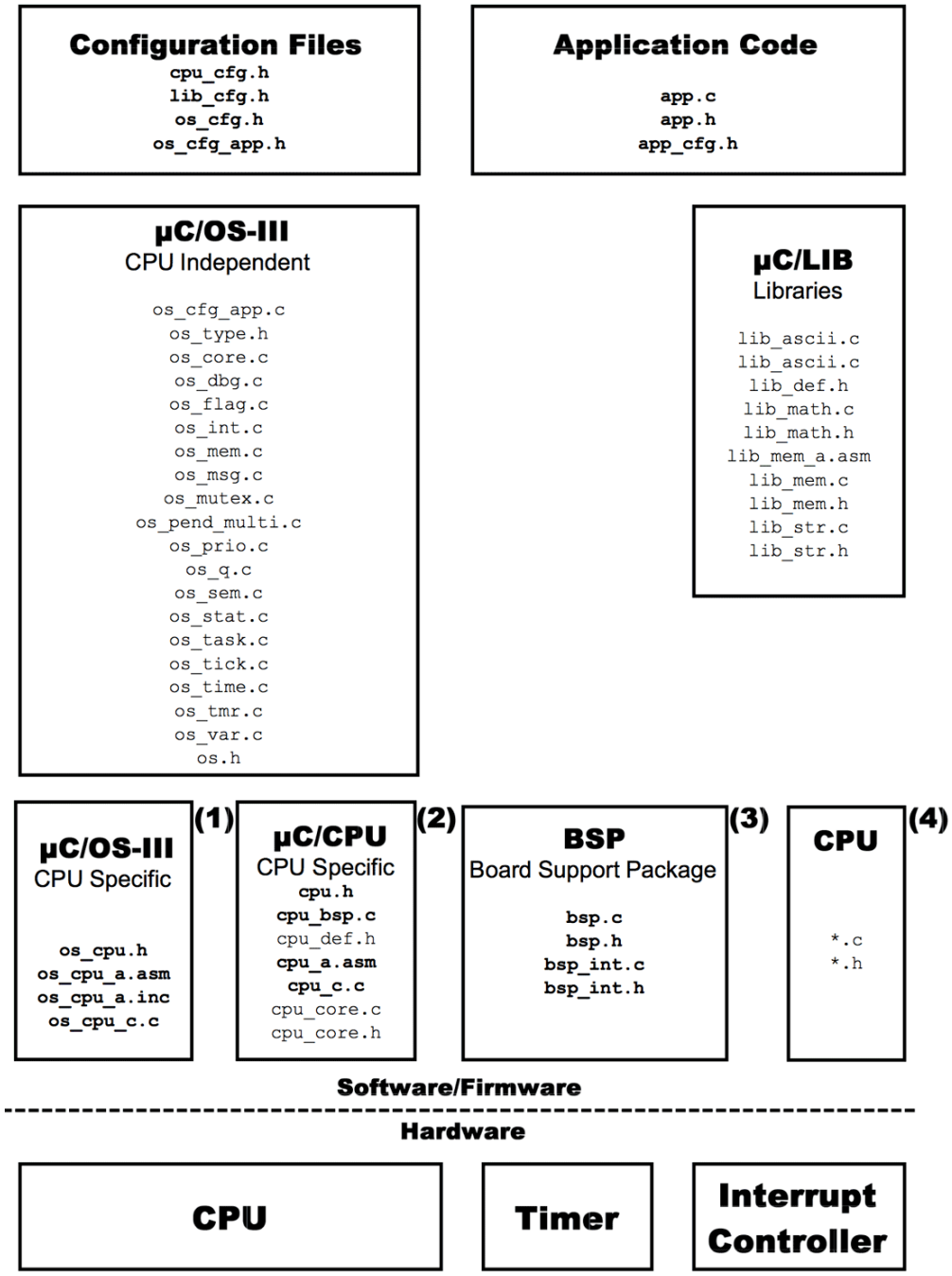


Figure - µC/OS-III architecture

(1) The port developer is responsible for providing the **µC/OS-III CPU Specific** portion. A µC/OS-III port consists of writing or changing the contents of four kernel-specific files:

os_cpu.h, os_cpu_a.asm, os_cpu_a.inc and os_cpu_c.c.

(2) A port also involves writing or changing the contents of two CPU specific files: cpu.h and cpu_a.asm. cpu_core.c is generally generic and should not require modifications.

(3) A Board Support Package (BSP) is generally necessary to interface µC/OS-III to a timer (which is used for the clock tick) and an interrupt controller.

(4) Some semiconductor manufacturers provide source and header files to access on-chip peripherals. These are contained in CPU/MCU specific files. You generally don't need to modify any of these and thus, you can use them as-is.

Porting µC/OS-III is quite straightforward once the subtleties of the target processor and the C compiler/assembler are understood. Depending on the processor, a port consists of writing or changing between 100 and 400 lines of code, which takes a few hours to a few days to accomplish. The easiest thing to do, however, is to modify an existing port from a processor that is similar to the one intended for use.

A µC/OS-III port looks very much like a µC/OS-II port. Since µC/OS-II was ported to well over 45 different CPU architectures it is easy to start from a µC/OS-II port. Converting a µC/OS-II port to µC/OS-III takes approximately an hour. The process is described in Migrating from uC-OS-II to uC-OS-III.

A port involves three aspects: CPU, OS and board-specific code. The board-specific code is often called a *Board Support Package* (BSP) and from µC/OS-III's point of view, requires very little.

In this section, we'll present the steps needed to do a port from scratch. Actually, you'll be starting from templates files that already contain placeholders for the code you'll need to insert.

The following is the layout for this section:

- Conventions
- µC/CPU Port Files

Copyright 2015 Micrium Inc.

- µC/OS-III Port Files
- BSP Files
- Testing a Port

Copyright 2015 Micrium Inc.

Conventions

µC/CPU and µC/OS-III are provided in source form and include template files (C and assembly language) which contain instructions about what code needs to be inserted and where. Specifically, you will need to search the source files for four dollar signs (i.e., $$$$) which are used as placeholders and replace those with the necessary code.

It is assumed that assembly language code use a file extension of `.asm`. Other assembler might require that the extension be `.s` or `.src`. If that's the case with your tools, simply name the assembly language files using the proper extension.

It is assumed that comments in an assembly language file starts with a semicolon, '`;`'.

In assembly language, there are a number of 'directives' that tell the assembler how to process certain pieces of information. Below is a list of such directives and their meaning. The assembler you use may use different syntax for these directives but overall, they should work and mean the same.

- The `PUBLIC` directive indicates that you are declaring a symbol to be globally available. In other words, it's public for all files to see.

- The `EXTERN` directive indicates that the definition of a symbol is found in another file (external to this file).

- The `CODE` directive indicates that what follows needs to be linked with the rest of your executable code. In other words, we are not declaring any variables.

- The `MACRO` directive is used to define an assembly language macro. A macro is basically a series of assembly language instructions that will be replacing the macro name. In other words, when the assembler sees the macro name being invoked in your code, it will replace the macro name with the instructions that are associated with the macro. Macros are useful to avoid repeating sequences of assembly language instructions.

- The `END` directive is generally the last assembly language statement in an assembly language file. The END directive should not appear at the end of a file that defines macros because macro files are generally included in other assembly language files.

Copyright 2015 Micrium Inc.

uC-CPU

About µC/CPU

µC/CPU is a module that provides CPU-specific functionality that is independent of µC/OS-III. Specifically, µC/CPU defines compiler and CPU dependent data types, the word width of the stack, whether the stack grows from high-to-low memory or from low-to-high memory, functions for disabling and enabling interrupts and more. Additional information about this module is provided in the µC/CPU User Manual.

The table below shows the name of µC/CPU files and where they should be placed on the computer used to develop a µC/OS-III-based application. The file names in **bold** are files you will need to create or modify for your own port.

File	Directory
cpu_bsp.c	\Micrium\Software\uC-CPU\BSP\Template\cpu_bsp.c
cpu_def.h	\Micrium\Software\uC-CPU\
cpu_cfg.h	\Micrium\Software\uC-CPU\CFG\Template
cpu_core.c	\Micrium\Software\uC-CPU\
cpu_core.h	\Micrium\Software\uC-CPU\
cpu.h	\Micrium\Software\uC-CPU\<processor>\<compiler>
cpu_c.c	\Micrium\Software\uC-CPU\<processor>\<compiler>
cpu_a.asm	\Micrium\Software\uC-CPU\<processor>\<compiler>

Table - µC/CPU files and directories

<processor> is the name of the processor that the cpu*.* files apply to.

<compiler> is the name of the toolchain (compiler, assembler, linker/locator) used. Each has its own directory because they may have different features that makes them different from one another.

Copyright 2015 Micrium Inc.

cpu_bsp.c

This file contains skeleton functions for `CPU_TS_TmrInit()`, `CPU_TS_TmrRd()` and other time stamp related functions. You can copy this file to your Board Support Package (BSP) directory, modify its content and add it to your build.

cpu_def.h

This file should not require any changes. `cpu_def.h` declares `#define` constants that are used by Micriμm software components.

cpu_cfg.h

This is a configuration file to be copied into the product directory and changed based on the options to exercise in µC/CPU. `cpu_cfg.h` is not considered a ‘port file’ but more an application specific file. However, it’s discussed here for completeness. The file contains `#define` constants that may need to be changed based on the desired use of µC/CPU.

CPU_CFG_NAME_EN

This `#define` determines whether you will be assigning a ‘name’ to the CPU port. This name can then be read by application code.

CPU_CFG_NAME_SIZE

This `#define` specifies the length of the ASCII string used to assign a name to the CPU.

CPU_CFG_TS_32_EN

This `#define` specifies whether 32-bit time stamps are available for this CPU. A 32-bit timestamp is typically the value of a free-running 32-bit counter that is used to make accurate time measurements. The application code can obtain the current value of this free-running timer at any point in time and use such value to determine when an event occurred or, measure the time difference between two events. The free-running counter is generally incremented at a fairly high rate, for example 1 MHz or more.

Copyright 2015 Micrium Inc.

CPU_CFG_TS_64_EN

This `#define` specifies whether 64-bit time stamps are available for this CPU. A 64-bit timestamp is typically the value of a free-running 64-bit counter (possibly made up by counting overflows of a 32-bit counter) that is used to make accurate time measurements. The application code can obtain the current value of this free-running timer at any point in time and use such value to determine when an event occurred or, measure the time difference between two events. The free-running counter is generally incremented at a fairly high rate, for example 100 MHz or more.

CPU_CFG_TS_TMR_SIZE

This `#define` specifies the size, in number of bytes, of a timestamp. A 32-bit timestamp is 4 bytes long while a 64-bit timestamp is 8 bytes long.

CPU_CFG_INT_DIS_MEAS_EN

This `#define` specifies whether extra code will be inserted to measure interrupt disable time when the code uses `CPU_CRITICAL_ENTER()` and `CPU_CRITICAL_EXIT()`. This extra code obviously adds additional interrupt latency because of the measurement artifacts.

CPU_CFG_INT_DIS_MEAS_OVRHD_NBR

This `#define` is used to account for the interrupt disable time measurement artifacts. The value should typically be 1.

CPU_CFG_LEAD_ZEROS_ASM_PRESENT

This `#define` specifies whether or not your processor offers assembly language instructions to count the number of consecutive zeros from the left most bit position of an integer.

CPU_CFG_TRAIL_ZEROS_ASM_PRESENT

This `#define` specifies whether or not your processor offers assembly language instructions to count the number of consecutive zeros from the right most bit position of an integer.

Copyright 2015 Micrium Inc.

cpu_core.c

This file is generic and does not need to be changed. However it must be included in all builds. `cpu_core.c` defines such functions as `CPU_Init()`, `CPU_CntLeadZeros()`, and code to measure maximum CPU interrupt disable time. A few functions are explained here since they are used in µC/OS-III-based applications.

CPU_Init()

`CPU_Init()` must be called before calling `OSInit()`.

CPU_CntLeadZeros()

`CPU_CntLeadZeros()` is used by the µC/OS-III scheduler to find the highest priority ready task (see Scheduling). `cpu_core.c` implements a count leading zeros in C. However, if the processor used provides a built-in instruction to count leading zeros, define `CPU_CFG_LEAD_ZEROS_ASM_PRESENT`, and replace this function by an assembly language equivalent (in `cpu_a.asm`). It is important to properly declare `CPU_CFG_DATA_SIZE` in `cpu.h` for this function to work.

CPU_TS_TmrFreqGet()

`CPU_TS_TmrFreqGet()` is a function that can be called by your application so it knows what the timestamp increment frequency is set to. In other words, if the timestamp timer is incremented at a rate of 1 MHz then this function would return `1000000`.

CPU_TS_TmrFreqSet()

`CPU_TS_TmrFreqSet()` is a function that needs to be called by the application code to notify µC/CPU about the increment frequency of the timer used for timestamps. In other words, if the timestamp timer is incremented at a rate of 1 MHz then your application code would need to call `CPU_TS_TmrFreqSet()` and pass `1000000`.

CPU_TS_Get32()

`CPU_TS_Get32()` is a function that returns a 32-bit timestamp. The macro `OS_TS_GET() (see os_cpu.h)` generally maps to this function.

Copyright 2015 Micrium Inc.

cpu_core.h

This header file is required by `cpu_core.c` to define function prototypes.

The table below shows the name of µC/CPU 'template' files you should use as a starting point should you decide to start a µC/CPU port from scratch. It's highly recommended that you copy these files to folders that matches the layout shown in this table. You would then edit these files to build up your own µC/CPU port files. Again, refer to the µC/CPU User's Manual (`uC-CPU-Manual.pdf`) found in `\Micrium\Software\uC-CPU\Doc`.

File	Directory
`cpu.h`	`\Micrium\Software\uC-CPU\Template`
`cpu_c.c`	`\Micrium\Software\uC-CPU\Template`
`cpu_a.asm`	`\Micrium\Software\uC-CPU\Template`

Table - µC/CPU template files

cpu.h

Many CPUs have different word lengths and `cpu.h` declares a series of type definitions that ensure portability. Specifically, we don't use the C data types `int`, `short`, `long`, `char`, *etc. at Micrium*. Instead, clearer data types are defined. Consult your compiler documentation to determine whether the standard declarations described below need to be changed for the CPU/compiler you are using. You should note that the `typedefs` below are not all grouped together in `cpu.h` and also, `cpu.h` contains additional comments about these data types.

Copyright 2015 Micrium Inc.

```
typedef           void        CPU_VOID;
          typedef  unsigned  char        CPU_CHAR;
          typedef  unsigned  char        CPU_BOOLEAN;
          typedef  unsigned  char        CPU_INT08U;
          typedef    signed  char        CPU_INT08S;
          typedef  unsigned  short       CPU_INT16U;
          typedef    signed  short       CPU_INT16S;
          typedef  unsigned  int         CPU_INT32U;
          typedef    signed  int         CPU_INT32S;
          typedef  unsigned  long  long  CPU_INT64U;
          typedef    signed  long  long  CPU_INT64S;
          typedef            float       CPU_FP32;
          typedef            double      CPU_FP64;
          typedef  volatile  CPU_INT08U  CPU_REG08;
          typedef  volatile  CPU_INT16U  CPU_REG16;
          typedef  volatile  CPU_INT32U  CPU_REG32;
          typedef  volatile  CPU_INT64U  CPU_REG64;
          typedef            void      (*CPU_FNCT_VOID)(void);
          typedef            void      (*CPU_FNCT_PTR )(void *);
          typedef  CPU_INT32U            CPU_ADDR;
          typedef  CPU_INT32U            CPU_DATA;
          typedef  CPU_DATA              CPU_ALIGN;
          typedef  CPU_ADDR              CPU_SIZE_T;
          typedef  CPU_INT32U            CPU_STK;             (1)
          typedef  CPU_ADDR              CPU_STK_SIZE;
          typedef  CPU_INT16U            CPU_ERR;
          typedef  CPU_INT32U            CPU_SR;              (2)
          typedef  CPU_INT32U            CPU_TS;              (3)
```

Listing - µC/CPU Data Types

(1) Especially important for µC/OS-III is the definition of the `CPU_STK` data type, which sets the width of a stack entry. Specifically, is the width of data pushed to and popped from the stack 8 bits, 16 bits, 32 bits or 64 bits?

(2) `CPU_SR` defines the data type for the processor's status register (SR) that generally holds the interrupt disable status.

(3) The `CPU_TS` is a time stamp used to determine when an operation occurred, or to measure the execution time of code.

`cpu.h` also declares macros to disable and enable interrupts: `CPU_CRITICAL_ENTER()` and `CPU_CRITICAL_EXIT()`, respectively. The documentation in the template file explains how to declare these macros.

`cpu.h` is also where you need to define configuration constants:

Copyright 2015 Micrium Inc.

CPU_CFG_ENDIAN_TYPE

This `#define` specifies whether your CPU is a little-endian machine or a big-endian machine. `cpu_def.h` offers the following choices:

```
CPU_ENDIAN_TYPE_BIG
CPU_ENDIAN_TYPE_LITTLE
```

CPU_CFG_ADDR_SIZE

This `#define` specifies the size of an address for the processor, in number of bytes. `cpu_def.h` offers the following choices:

```
CPU_WORD_SIZE_08
CPU_WORD_SIZE_16
CPU_WORD_SIZE_32
CPU_WORD_SIZE_64
```

CPU_CFG_DATA_SIZE

This `#define` specifies the ‘natural’ data width of the processor, in number of bytes. `cpu_def.h` offers the following choices:

```
CPU_WORD_SIZE_08
CPU_WORD_SIZE_16
CPU_WORD_SIZE_32
CPU_WORD_SIZE_64
```

CPU_DATA_SIZE_MAX

This `#define` specifies the maximum word size of the processor, in number of bytes. `cpu_def.h` offers the following choices:

```
CPU_WORD_SIZE_08
CPU_WORD_SIZE_16
CPU_WORD_SIZE_32
CPU_WORD_SIZE_64
```

Copyright 2015 Micrium Inc.

CPU_CFG_STK_GROWTH

This #define specifies whether the stack grows from high to low memory or from low to high memory addresses. cpu_def.h offers the following choices:

```
CPU_STK_GROWTH_LO_TO_HI
CPU_STK_GROWTH_HI_TO_LO
```

CPU_CFG_CRITICAL_METHOD

This #define establishes how interrupts will be disabled when processing critical sections. Specifically, will we simply disable interrupts when entering a critical section, irrespective of whether or not interrupts were already disabled? Will we save the status of the interrupt disable state before we disable interrupts? cpu_def.h offers the following choices:

```
CPU_CRITICAL_METHOD_INT_DIS_EN
CPU_CRITICAL_METHOD_STATUS_STK
CPU_CRITICAL_METHOD_STATUS_LOCAL
```

cpu.h also declares function prototypes for a number of functions found in either cpu_c.c or cpu_a.asm.

cpu_c.c

This is an optional file containing CPU-specific functions to set the interrupt controller, timer prescalers, and more. Most implementations will not contain this file.

cpu_a.asm

This file contains assembly language code to implement such functions as disabling and enabling interrupts, a more efficient count leading zeros function, and more. At a minimum, this file should implement CPU_SR_Save() and CPU_SR_Restore().

Copyright 2015 Micrium Inc.

CPU_SR_Save()

`CPU_SR_Save()` reads the current value of the CPU status register where the current interrupt disable flag resides and returns this value to the caller. However, before returning, `CPU_SR_Save()` must disable all interrupts. `CPU_SR_Save()` is actually called by the macro `CPU_CRITICAL_ENTER()`.

CPU_SR_Restore()

`CPU_SR_Restore()` restores the CPU's status register to a previously saved value. `CPU_SR_Restore()` is called from the macro `CPU_CRITICAL_EXIT()`.

Copyright 2015 Micrium Inc.

uC-OS-III Port

Porting µC/OS-III

The table following the figure below below shows the name of µC/OS-III files and where they are typically found.

For the purpose of demonstrating how to do a port, we will assume the generic 32-bit processor as described in Context Switching and shown in the figure below.

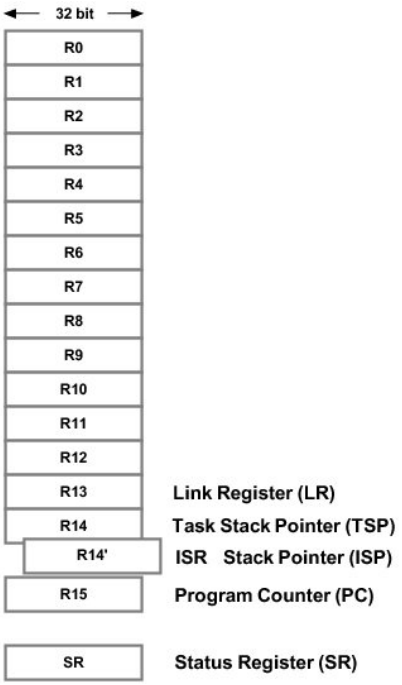


Figure - Generic 32-bit CPU

Our generic CPU contains 16 integer registers (R0 to R15), a separate ISR stack pointer, and a separate status register (SR). Every register is 32 bits wide and each of the 16 integer registers can hold either data or an address. The return address of a function call is placed in the Link Register (`LR`). The program counter (or instruction pointer) is R15 and there are two separate stack pointers labeled R14 and R14'. R14 represents a task stack pointer (TSP), and R14' represents an ISR stack pointer (ISP). The CPU automatically switches to the ISR stack when servicing an exception or interrupt. The task stack is accessible from an ISR (i.e., we can push

Copyright 2015 Micrium Inc.

and pop elements onto the task stack when in an ISR), and the interrupt stack is also accessible from a task. The Status Register (SR) contains the interrupt mask as well as various status such as the Carry, Zero, Sign, Overflow, Parity, etc.

File	Directory
bsp_os.c bsp_os.h bsp_os_a.asm	\Micrium\Software\EvalBoards\<manufacturer>\<board>\BSP\OS\uCOS-III\
os_cpu.h	\Micrium\Software\uCOS-III\Ports\<processor>\<compiler>\
os_cpu_a.asm	\Micrium\Software\uCOS-III\Ports\<processor>\<compiler>\
os_cpu_a.inc	\Micrium\Software\uCOS-III\Ports\<processor>\<compiler>
os_cpu_c.c	\Micrium\Software\uCOS-III\Ports\<processor>\<compiler>\

Table - µC/OS-III files and directories

Here, `<processor>` is the name of the processor that the `os_cpu*.*` files apply to, and `<compiler>` is the name of the compiler that these files assume because of the different assembly language directives that different toolchain uses.

The table below shows where you can find template files that will help you create a µC/OS-III port from scratch. You would simply copy these files in a folder specific to your processor/compiler as shown in the table above and then change the contents of these files per your processor/compiler.

File	Directory
os_cpu.h	\Micrium\Software\uCOS-III\Ports\Template\
os_cpu_a.asm	\Micrium\Software\uCOS-III\Ports\Template\
os_cpu_a.inc	\Micrium\Software\uCOS-III\Ports\Template\
os_cpu_c.c	\Micrium\Software\uCOS-III\Ports\Template\

Table - µC/OS-III template files

Copyright 2015 Micrium Inc.

bsp_os.c, bsp_os_a.asm and bsp_os.h — Periodic Tick

Board Support Package (BSP) specific code is generally needed to provide µC/OS-III with a periodic time source. This is typically obtained from a hardware timer that is configured to generate a ‘tick rate’ between 10 and 1000 Hz. By convention, we decided to call these files bsp_os.c, bsp_os_a.asm (optional) and bsp_os.h. These files, part of the BSP, typically contain code for just a couple of functions: `BSP_OS_TickInit()` and `BSP_OS_TickISR()`.

BSP_OS_TickInit()

This function must be called by the first task that executes under µC/OS-III and after you called `CPU_Init()`. Alternatively, you can place `BSP_OS_TickInit()` in `os_cpu_c.c` depending on whether or not the tick ISR is generic for the CPU architecture you are using. In other words, if the CPU or MCU has a dedicated timer that can be assigned for the tick ISR so that it’s the same, irrespective of the target application then `BSP_OS_TickInit()` can be placed in `os_cpu_c.c`. The pseudo-code for this function is shown in the listing below.

```
void  BSP_OS_TickInit (CPU_INT32U  freq)                                        (1)
{
    Install the interrupt vector for the timer used to generate tick interrupts;   (2)
    Configure the timer to generate interrupts at ‘freq’ Hz;                       (3)
    Enable timer interrupts;                                                       (4)
}
```

Listing - BSP_OS_TickInit() Pseudo Code

BSP_OS_TickISR()

If ISRs are implemented in assembly language this function could be placed in a file called `bsp_os_a.asm` (or other filename extension as needed by your assembler). Alternatively, you can place `BSP_OS_TickISR()` in `os_cpu_a.asm` depending on whether or not the tick ISR is generic for the CPU and, whether it needs to be implemented in assembly language. In other words, if the CPU or MCU has a dedicated timer that can be assigned for the tick ISR so that it’s the same, irrespective of the target application then `BSP_OS_TickISR()` can be placed in `os_cpu_a.asm`. The pseudo code for this function is shown below (the C-like code needs to be implemented in assembly language). You should note that all ISRs should be modeled after `BSP_OS_TickISR()`.

Copyright 2015 Micrium Inc.

```
BSP_OS_TickISR:                                                  (1)
   OS_CTX_SAVE                                                   (2)
   Disable Interrupts;                                           (3)
   OSIntNestingCtr++;                                            (4)
   if (OSIntNestingCtr == 1) {                                   (5)
       OSTCBCurPtr->StkPtr = SP;
   }
   Clear tick interrupt;                                         (6)
   OSTimeTick();                                                 (7)
   OSIntExit();                                                  (8)
   OS_CTX_RESTORE                                                (9)
   Return from Interrupt/Exception;                             (10)
```

Listing - BSP_OS_TickISR() Pseudo Code

(1) `BSP_OS_TickISR()` is generally invoked automatically by the interrupt controller when the tick interrupt occurs. Assuming again our generic 32-bit CPU, it's assumed here that the `SR` and `PC` of the interrupted task are pushed automatically onto the stack of the interrupted task.

(2) Again, `OS_CTX_SAVE` saves the CPU context onto the current task's stack. For our generic 32-bit CPU, `OS_CTX_SAVE` would push `R0` through `R13` onto the stack, in that order.

(3) Interrupts should be disabled here. On some processors, interrupts are automatically disabled when the processor accepts the interrupt. Some processors support multiple interrupt levels. In fact, some interrupts are allowed to make kernel calls while others are not. Typically, interrupts that do not make kernel calls (called Non-Kernel Aware Interrupts) would generally be high priority interrupts and kernel aware interrupts would all be grouped (in priority) below these. For example, if a processor has `16` different interrupt levels and level `0` is the lowest priority interrupt then, all kernel aware interrupts would be assigned from 0 to some number `N` (let's say `12`) and `N+1` to `15` would be assigned to be non-kernel aware interrupts.

(4) `BSP_OS_TickISR()` then needs to increment the interrupt nesting counter. This tells µC/OS-III that the code is servicing an interrupt. The nesting counter indicates how many levels of interrupts we are currently servicing (in case the application supports nested interrupts).

(5) If this interrupt interrupts a task then we need to save the stack pointer of that task into the `OS_TCB` of that task.

Copyright 2015 Micrium Inc.

(6) You need to clear the interrupting device so that it doesn't re-issue the same interrupt upon returning from interrupts. This can be done here or, in the device handler (see below).

(7) At this point, `BSP_OS_TickISR()` calls `OSTimeTick()` which is responsible for notifying the tick task that a tick occurred.

If you model your ISR like `BSP_OS_TickISR()` then you would call your own C function to service the interrupting device.

(8) `OSIntExit()` is then called at the end of the ISR to notify µC/OS-III that you are done processing the ISR. µC/OS-III decrements the nesting counter and if `OSIntNestingCtr` reaches `0`, µC/OS-III knows it's returning to task level code. So, if the ISR made a more important task ready-to-run (more important than the interrupted task), µC/OS-III will context switch to that task instead of returning to the interrupted task.

(9) If the interrupted task is still the most important task then `OSIntExit()` returns and the ISR will need to restore the saved registers. `OS_CTX_RESTORE` does just that. For our generic 32-bit CPU, `OS_CTX_RESTORE` would pop CPU registers `R13` through `R0` from the stack, in that order.

(10) Finally, the Return from Interrupt/Exception restores the Program Counter (`PC`) and the Status Register (`SR`) in a single instruction. At this point, the interrupted task will resume execution, exactly where it was interrupted.

It is actually possible to simplify the code for `BSP_OS_TickISR()` or any of your ISRs. Notice that the code at the beginning and end of the ISR is common for all ISRs. Because of that, it's possible to create two assembly language macros, `OS_ISR_ENTER` and `OS_ISR_EXIT` in `os_cpu_a.inc`. The new `BSP_OS_TickISR()` code would now look as shown below:

Copyright 2015 Micrium Inc.

```
BSP_OS_TickISR:
    OS_ISR_ENTER
    Clear tick interrupt;
    OSTimeTick();
    OS_ISR_EXIT
```

Listing - BSP_OS_TickISR() Pseudo Code using the OS_ISR_ENTER and OS_ISR_EXIT macros

bsp_os.c, bsp_os_a.asm and bsp_os.h — Dynamic Tick

Board Support Package (BSP) specific code is needed to provide µC/OS-III with a dynamic time source. To use the dynamic ticking feature, a programmable timer is necessary. This timer has to be precise enough to guarantee the execution of the Tick task at least 1000 times per second. In other words, the frequency of the dynamic tick timer must be an integer multiple of `OS_CFG_TICK_RATE_HZ` to properly manage single `OS_TICK` delays. When the `OS_CFG_DYN_TICK_EN` option is set to `DEF_ENABLED` in `os_cfg.h`, the port developer needs to implement the `BSP_OS_DynTickInit()`, `BSP_OS_DynTickHandler()`, `BSP_OS_TickGet()` and `BSP_OS_TickNextSet()` functions. Otherwise, the Dynamic Tick feature will not function properly.

The following sections contain descriptions of the Dynamic Tick functions.

BSP_OS_DynTickInit()

Like the periodic tick, this function must be called by the first task that executes under µC/OS-III and after you called `CPU_Init()`. The pseudo-code for this function is similar to the code for the Periodic tick and is shown in the listing below.

```
void  BSP_OS_DynTickInit (void)
{
    Install the interrupt vector (BSP_OS_DynTickHandler) for the timer used to generate tick interrupts;
    Configure the timer to generate interrupts but deactivate its period;
    Enable timer interrupts;
}
```

Listing - BSP_OS_DynTickInit() Pseudo Code

Copyright 2015 Micrium Inc.

BSP_OS_DynTickHandler()

The Dynamic tick ISR is executed when a dynamic tick interrupt occurs. This should occur after at least Tick Step OS ticks have elapsed. The ISR must inform µC/OS-III, using the `OSTimeDynTick()` function, about the effective number of OS ticks that has elapsed since the last time a Dynamic Tick interrupt occurred. The Dynamic Tick timer is programmed using the `BSP_OS_DynTickNextSet()` function described below.

```
void  BSP_OS_DynTickHandler (void)
{
    OSIntEnter();
    Clear tick interrupt;
    OSTimeDynTick(Elapsed OS Ticks since last interrupt);
    Save The Last OSTickCtr;
    OSIntExit();
}
```

Listing - BSP_OS_DynTickHandler() Pseudo code

BSP_OS_TickGet()

This particular function is used by µC/OS-III to get the OS time (`OSTickCtr`) as if it were incremented `OS_CFG_TICK_RATE_HZ` times per second.

`BSP_OS_TickGet()` exists only because the OS time counter isn't incremented periodically. Indeed, while µC/OS-III is waiting for Tick Step ticks to elapse, an interrupt could occur that would resume a previously interrupted task. That task, in turn, could try to delay itself. Now, even though the OS time counter hasn't been updated since µC/OS-III is still waiting for Tick Step ticks to elapse, time has **still** elapsed. This function is used to measure how much time has elapsed since `OSTickCtr` was last updated. `BSP_OS_TickGet()`'s pseudo-code is listed below:

```
OS_TICK  BSP_OS_TickGet (void)
{
    return The Last OSTickCtr + The Timer Counter Converted to OS Tick units;  /* given the fact that
the Timer isn't OS_CFG_TICK_RATE_HZ times per second */
}
```

Listing - BSP_OS_TickGet() Pseudo code

Copyright 2015 Micrium Inc.

BSP_OS_TickNextSet()

As explained in The Tick Task, this function gets called when a task needs to delay its execution, when a task waits with a timeout on a kernel object and every time the Tick task is run. The argument passed to this function is the number of OS ticks that need to elapse before the Tick task is run again. This value is called the Tick Step and represents the minimum delay needed to wake up a task present in either the timeout tick list or the delay tick list.

The operating frequency of the dynamic tick timer can be higher than `OS_CFG_TICK_RATE_HZ`. Which means that once converted to a dynamic tick count, a delay in `OS_TICKs` could overflow the counter used by the dynamic tick timer. For example, suppose the dynamic tick timer uses a 16 bit counter and operates at 10 kHz and that `OS_CFG_TICK_RATE_HZ` is set to 1000. A delay of 7 seconds, or 7000 `OS_TICKs` would need 7000*(10000/1000) or 70000 dynamic timer ticks. This is because one `OS_TICK` is worth 10 dynamic timer ticks since the dynamic tick timer operates at a frequency that is 10 times faster than `OS_CFG_TICK_RATE_HZ`. In this case, 70000 is effectively higher than the maximum value that the 16 bit counter can contain. `BSP_OS_TickNextSet()` would then set the dynamic tick timer to interrupt in the maximum number of `OS_TICKs` it can contain. Doing so, `BSP_OS_TickNextSet()` would set the number of effective OS ticks that will elapse to 6553. This is because even though the maximum value of a 16 bit timer is 65535, once this value is converted to `OS_TICKS`, it is not even. One needs to find the maximum number of `OS_TICKs` the dynamic tick timer counter can contain without overflowing and without yielding an odd `OS_TICK` count equivalent. In this example, its 65530 since divided by 10 it gives an even 6553 `OS_TICKs`.

`BSP_OS_TickNextSet()`'s pseudo-code is listed below:

```
OS_TICK  BSP_OS_TickNextSet (OS_TICK  ticks)
{
    If needed, stop the Timer;

    Convert the needed "ticks" in dynamic timer ticks;
    Verify and adjust the dynamic timer tick so that it does not overflow and still expresses an even
number of OS_TICKs;
    Set the Timer to interrupt after the calculated dynamic timer ticks have elapsed;
    Save the number of effective OS ticks that will elapse before next interrupt;

    If stopped, restart the Timer;

    Return the the number of effective OS ticks that will elapse before next interrupt;
}
```

Listing - BSP_OS_TickNextSet() Pseudo code

Copyright 2015 Micrium Inc.

os_cpu.h

OS_TASK_SW()

`OS_TASK_SW()` is a macro that is called by `OSSched()` to perform a task-level context switch. The macro can translate directly to a call to `OSCtxSw()`, trigger a software interrupt, or a `TRAP`. If a software interrupt or `TRAP` is used then you would most likely need to add the address of `OSCtxSw()` in the interrupt vector table. The choice depends on the CPU architecture.

OS_TS_GET()

`OS_TS_GET()` is a macro that obtains the current time stamp. It is expected that the time stamp is type `CPU_TS`, which is typically declared as at least a 32-bit value.

OSCtxSw(), OSIntCtxSw() and OSStartHighRdy()

`os_cpu.h` declares function prototypes for `OSCtxSw()`, `OSIntCtxSw()`, `OSStartHighRdy()` and possibly other functions required by the port.

os_cpu_c.c

The functions are described in µC-OS-III API Reference. `os_cpu_c.c` can declare any functions needed by the port, however the functions described below are mandatory. These functions are already implemented in the template file, but those can certainly be extended as needed. You should not have to change this file unless you have specific requirements.

OSIdleTaskHook()

This function is called repeatedly when µC/OS-III does not have any task ready-to-run. The port implemented might choose to put the processor in low power mode if the product being designed is battery operated. However, it would be preferable to defer this choice to the application level. You can do this by putting the processor in low power mode in a function called `App_OS_IdleTaskHook()` and let the application decide whether or not it is appropriate to place the processor in low power mode. The template file contains the following code:

Copyright 2015 Micrium Inc.

```
void  OSIdleTaskHook (void)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppIdleTaskHookPtr != (OS_APP_HOOK_VOID)0) {
        (*OS_AppIdleTaskHookPtr)();
    }
#endif
}
```

Listing - Typical OSIdleTaskHook()

OSInitHook()

This function is called by `OSInit()` at the very beginning of `OSInit()`. This is done to allow the port implemented to add functionality to the port while hiding the details from the µC/OS-III user. For one thing, the port implementer could setup an ISR stack in `OSInitHook()`. The template file contains the following code:

```
void  OSInitHook (void)
{
}
```

Listing - Typical OSInitHook()

OSRedzoneHitHook()

If Redzone Stack Checking is enabled (`OS_CFG_TASK_STK_REDZONE_EN` set to `DEF_ENABLED` in is `os_cfg.h`), this function is called when µC/OS-III determines that a task's stack has overflowed its Redzone. The function calls an application hook, if defined. If not, it calls a software exception. The application hook could try to fix the stack, report an error or simply call the software exception.

```
void  OSRedzoneHitHook (OS_TCB  *p_tcb)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppRedzoneHitHookPtr != (OS_APP_HOOK_TCB)0) {
        (*OS_AppRedzoneHitHookPtr)(p_tcb);
    }
#endif
    (void)p_tcb;
    CPU_SW_EXCEPTION(;);
}
```

Listing - Typical OSRedzoneHitHook()

Copyright 2015 Micrium Inc.

OSStatTaskHook()

This function is called when the statistic task executes. This hook allows the port developer the opportunity to add his or her own statistics. The template file contains the following code:

```
void  OSStatTaskHook (void)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppStatTaskHookPtr != (OS_APP_HOOK_VOID)0) {
        (*OS_AppStatTaskHookPtr)();
    }
#endif
}
```

Listing - Typical OSStatTaskHook()

OSTaskCreateHook()

This function is called by `OSTaskCreate()` and is passed the address of the `OS_TCB` of the newly created task. `OSTaskCreateHook()` is called by `OSTaskCreate()` after initializing the `OS_TCB` fields and setting up the stack frame for the task. The template file contains the following code:

```
void  OSTaskCreateHook (OS_TCB  *p_tcb)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppTaskCreateHookPtr != (OS_APP_HOOK_TCB)0) {
        (*OS_AppTaskCreateHookPtr)(p_tcb);
    }
#else
    (void)p_tcb;
}
```

Listing - Typical OSTaskCreateHook()

OSTaskDelHook()

This function is called by `OSTaskDel()` after the task to delete has been removed either from the ready list or a wait list. The template file contains the following code:

```
void  OSTaskDelHook (OS_TCB  *p_tcb)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppTaskDelHookPtr != (OS_APP_HOOK_TCB)0) {
        (*OS_AppTaskDelHookPtr)(p_tcb);
    }
#else
    (void)p_tcb;
#endif
}
```

Listing - Typical OSTaskDelHook()

OSTaskReturnHook()

This function is called by `OS_TaskReturn()` if the user accidentally returns from the task code. The template file contains the following code:

```
void  OSTaskReturnHook (OS_TCB  *p_tcb)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppTaskReturnHookPtr != (OS_APP_HOOK_TCB)0) {
        (*OS_AppTaskReturnHookPtr)(p_tcb);
    }
#else
    (void)p_tcb;
}
```

Listing - Typical OSTaskReturnHook()

OSTaskStkInit()

`OSTaskStkInit()` is called by `OSTaskCreate()` and is one of the most difficult port functions to create because it establishes the stack frame of every task created. The template file contains the following code:

```
CPU_STK  *OSTaskStkInit (OS_TASK_PTR     p_task,
                         void           *p_arg,
                         CPU_STK        *p_stk_base,
                         CPU_STK        *p_stk_limit,
                         CPU_STK_SIZE    stk_size,
                         OS_OPT          opt)
{
    CPU_STK  *p_stk;


    (void)opt;

    p_stk    = &p_stk_base[stk_size];              (1)

    *--p_stk = (CPU_STK)0x00000000u;               (2)
    *--p_stk = (CPU_STK)p_task;                    (3)
    *--p_stk = (CPU_STK)p_arg;                     (4)
    *--p_stk = (CPU_STK)0x01010101u;               (5)
    *--p_stk = (CPU_STK)0x02020202u;
    *--p_stk = (CPU_STK)0x03030303u;
    *--p_stk = (CPU_STK)0x04040404u;
    *--p_stk = (CPU_STK)0x05050505u;
    *--p_stk = (CPU_STK)0x06060606u;
    *--p_stk = (CPU_STK)0x07070707u;
    *--p_stk = (CPU_STK)0x08080808u;
    *--p_stk = (CPU_STK)0x09090909u;
    *--p_stk = (CPU_STK)0x10101010u;
    *--p_stk = (CPU_STK)0x11111111u;
    *--p_stk = (CPU_STK)0x12121212u;
    *--p_stk = (CPU_STK)OS_TaskReturn;             (6)

    return (p_stk);                                (7)
}
```

Listing - Generic OSTaskStkInit()

(1) You need to initialize the top-of-stack. For our ‘generic 32-bit CPU, the top-of-stack (TOS) points at one location beyond the area reserved for the stack. This is because we will decrement the TOS pointer before storing a value into the stack.

If the stack for the processor grew from low memory to high memory, most likely you would have setup the TOS to point at the base of the memory or, `&p_stk_base[0]`.

(2) Since we are simulating an interrupt and the stacking of registers in the same order as an ISR would place them on the stack, we start by putting the `SR` (Status Register, also called the Program Status Word) of the CPU onto the stack first.

Also, the value stored at this location must be such that, once restored into the CPU, the `SR` must enable ALL interrupts. Here we assumed that a value of `0x00000000` would do this. However, you need to check with the processor you are using to find out how this works on that processor.

Copyright 2015 Micrium Inc.

(3) The address of the task code is then placed onto the next stack location. This way, when you perform a return from interrupt (or exception) instruction the PC will automatically be loaded with the address of the task to run.

(4) You should recall that a task is passed an argument, `p_arg`. `p_arg` is a pointer to some user define storage or function and its use is application specific. In the assumptions above, we indicated that a function called with a single argument gets this argument passed in `R0`. You will need to check the compiler documentation to determine where ‘ `p_arg`’ is placed for your processor.

(5) The remaining registers are placed onto the stack. You will notice that we initialized the value of those registers with a hexadecimal number that corresponds to the register number. In other words, `R12` should have the value `0x12121212` when the task starts, `R11` should have the value `0x11111111` when the task starts and so on. This makes it easy to determine whether the stack frame was setup properly when you test the port. You would simply look at the register contents with a debugger and confirm that all registers have the proper values.

(6) Here we place the return address of the task into the location where the Link Register (`LR`) will be retrieved from. In this case, we force the return address to actually be `OS_TaskReturn()` allowing µC/OS-III to catch a task that is attempting to return. You should recall that this is not allowed with µC/OS-III.

(7) `OSTaskStkInit()` needs to return the new top-of-stack location. In this case, the top-of-stack points at the last element placed onto the stack.

The figure below shows how the stack frame looks like just before the function returns. `OSTaskCreate()` will actually save the new top-of-stack (`p_stk`) into the `OS_TCB` of the task being created.

Copyright 2015 Micrium Inc.

Figure - Stack frame created by OSTaskStkInit()

OSTaskSwHook()

The typical code for µC/OS-III's context switch hook is shown below. What `OSTaskSwHook()` does is highly dependent on a number of configuration options.

```
void  OSTaskSwHook (void)
{
#if OS_CFG_TASK_PROFILE_EN > 0u
    CPU_TS  ts;
#endif
#ifdef  CPU_CFG_INT_DIS_MEAS_EN
    CPU_TS  int_dis_time;
#endif


#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppTaskSwHookPtr != (OS_APP_HOOK_VOID)0) {                       (1)
        (*OS_AppTaskSwHookPtr)();
    }
#endif

#if OS_CFG_TASK_PROFILE_EN > 0u
    ts = OS_TS_GET();                                                       (2)
    if (OSTCBCurPtr != OSTCBHighRdyPtr) {
        OSTCBCurPtr->CyclesDelta  = ts - OSTCBCurPtr->CyclesStart;
        OSTCBCurPtr->CyclesTotal += (OS_CYCLES)OSTCBCurPtr->CyclesDelta;
    }

    OSTCBHighRdyPtr->CyclesStart = ts;
#endif

#ifdef  CPU_CFG_INT_DIS_MEAS_EN
    int_dis_time = CPU_IntDisMeasMaxCurReset();                             (3)
    if (OSTCBCurPtr->IntDisTimeMax < int_dis_time) {
        OSTCBCurPtr->IntDisTimeMax = int_dis_time;
    }
#endif

#if OS_CFG_SCHED_LOCK_TIME_MEAS_EN > 0u
    if (OSTCBCurPtr->SchedLockTimeMax < OSSchedLockTimeMaxCur) {            (4)
        OSTCBCurPtr->SchedLockTimeMax = OSSchedLockTimeMaxCur;
    }
    OSSchedLockTimeMaxCur = (CPU_TS)0;
#endif

#if (OS_CFG_TASK_STK_REDZONE_EN == DEF_ENABLED)
    stk_status = OSTaskStkRedzoneChk(DEF_NULL);                             (5)
    if (stk_status != DEF_OK) {
        OSRedzoneHitHook(OSTCBCurPtr);
    }
#endif
}
```

Listing - Typical OSTaskSwHook()

(1) If the application code defined a hook function to be called during a context switch then this function is called first. You application hook function can assume that `OSTCBCurPtr` points to the `OS_TCB` of the task being switched out while `OSTCBHighRdyPtr` points to the `OS_TCB` of the task being switched in.

(2) `OSTaskSwHook()` then computes the amount of time the current task ran. However, this includes the execution time of all the ISRs that happened while the task was running.

We then take a timestamp to mark the beginning of the task being switched in.

(3) `OSTaskSwHook()` then stores the highest interrupt disable time into the `OS_TCB` of the task being switched out. This allows a debugger or µC/Probe to display maximum interrupt disable time on a per-task basis.

(4) `OSTaskSwHook()` then captures the highest scheduler lock time and stores that in the `OS_TCB` of the task being switched out.

(5) If Redzone Stack Checking is enabled, the hook checks to see if the task that is about to be switched out has overflowed its stack.

OSTimeTickHook()

This function is called by `OSTimeTick()` and is called before any other code is executed in `OSTimeTick()`. The template file contains the following code. If the application code defines an application hook function then it is called as shown.

```
 void  OSTimeTickHook (void)
{
#if OS_CFG_APP_HOOKS_EN > 0u
    if (OS_AppTimeTickHookPtr != (OS_APP_HOOK_VOID)0) {
        (*OS_AppTimeTickHookPtr)();
    }
#endif
}
```

Listing - ypical OSTimeTickHook()

os_cpu_a.asm

This file contains the implementation of the following assembly language functions:

OSStartHighRdy()

This function is called by `OSStart()` to start multitasking. `OSStart()` will have determined the highest priority task (`OSTCBHighRdyPtr` will point to the `OS_TCB` of that task) that was created prior to calling `OSStart()` and will start executing that task. The pseudo code for this function is shown below (the C-like code needs to be implemented in assembly language):

Copyright 2015 Micrium Inc.

```
OSStartHighRdy:
    OSTaskSwHook();
    SP = OSTCBHighRdyPtr->StkPtr;                       (1)
    OS_CTX_RESTORE                                      (2)
    Return from Interrupt/Exception;                    (3)
```

Listing - OSStartHighRdy() Pseudo Code

(1) The Stack Pointer (`SP`) for the first task to execute is retrieved from the `OS_TCB` of the highest priority task that was created prior to calling `OSStart()`. The figure below shows the stack frame as pointed to by `OSTCBHighRdy->StkPtr`.

(2) `OS_CTX_RESTORE` is a macro (see `os_cpu_a.inc`) that restores the context of the CPU (`R0` through `R13`) from the new task's stack.

(3) The Return from Interrupt/Exception restores the Program Counter (`PC`) and the Status Register (`SR`) in a single instruction. At this point, the task will start executing. In fact, the task will think it was called by another function and thus, will receive '`p_arg`' as its argument. Of course, the task *must not* return.

Copyright 2015 Micrium Inc.

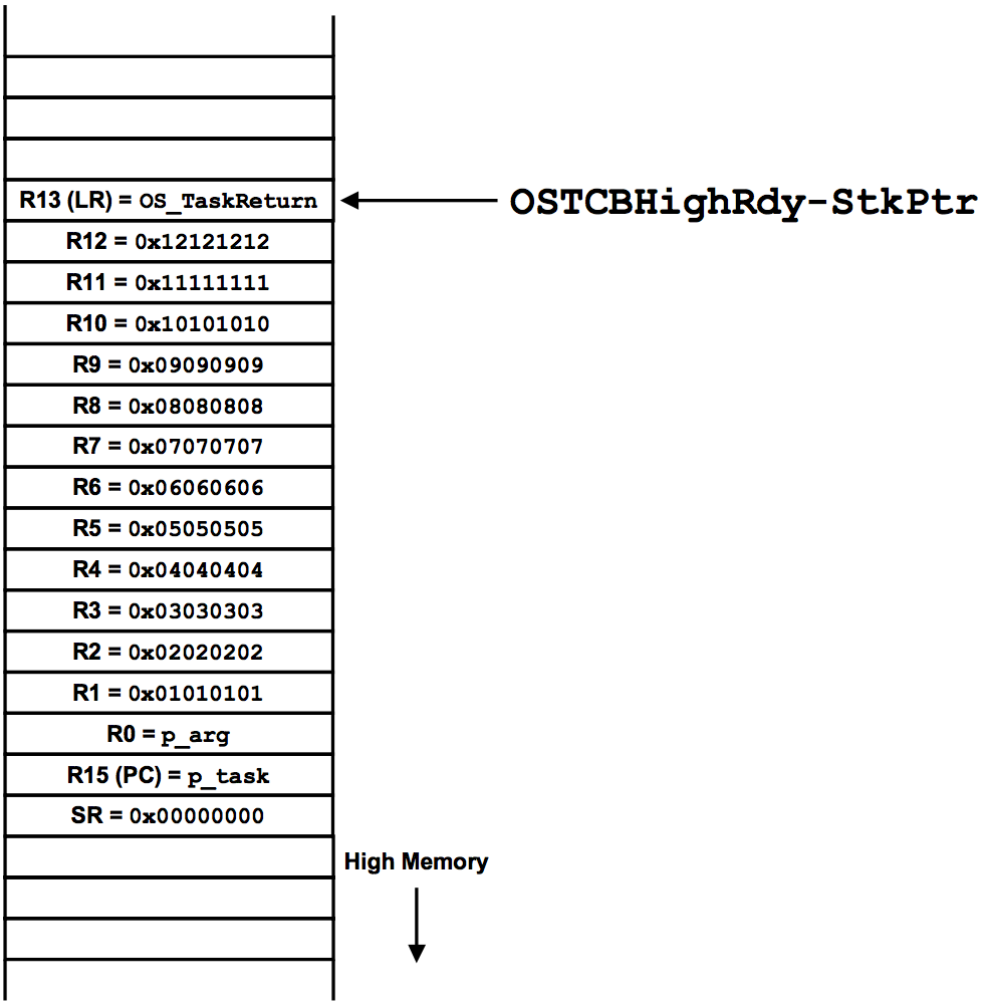


Figure - Stack Frame pointed to by OSTCBHighRdy->StkPtr

OSCtxSw()

This function implements the task level context switch which is invoked by the `OS_TASK_SW()` macro declared in `os_cpu.h`. The pseudo code for this function is shown below (the C-like code needs to be implemented in assembly language). You should also refer to Chapter 8.

```
OSCtxSw:                                                                (1)
    OS_CTX_SAVE                                                         (2)
    OSTCBCurPtr->StkPtr = SP;                                           (3)
    OSTaskSwHook();
    OSPrioCur   = OSPrioHighRdy;
    OSTCBCurPtr = OSTCBHighRdyPtr;
    SP          = OSTCBCurPtr->StkPtr;                                  (4)
    OS_CTX_RESTORE                                                      (5)
    Return from Interrupt/Exception;                                    (6)
```

Listing - OSCtxSw() Pseudo Code

(1) `OSCtxSw()` is invoked by `OS_TASK_SW()` which is typically implemented as a software interrupt instruction or, a trap instruction. These types of instructions generally simulate the behavior of an interrupt, but is synchronous with the code. `OSCtxSw()` is thus the entry point for this instruction. In other words, if a software interrupt or TRAP is used then you would most likely need to add the address of `OSCtxSw()` in the interrupt vector table.

(2) `OS_CTX_SAVE` is a macro (see `os_cpu_a.inc`) that saves the CPU context onto the current task's stack. For our generic 32-bit CPU, `OS_CTX_SAVE` would push `R0` through `R13` onto the stack, in that order.

(3) `OSCtxSw()` then needs to save the current top-of-stack pointer (i.e. `R14` or `SP`) into the `OS_TCB` of the current task.

(4) The stack pointer for the new task is retrieved from the `OS_TCB` of the new current task.

(5) `OS_CTX_RESTORE` is a macro (see `os_cpu_a.inc`) that restores the context of the CPU from the new task's stack. For our generic 32-bit CPU, `OS_CTX_RESTORE` would pop CPU registers `R13` through `R0` from the stack, in that order.

(6) The Return from Interrupt/Exception restores the Program Counter (`PC`) and the Status Register (`SR`) in a single instruction. At this point, the new task will resume execution, exactly where it was preempted.

Copyright 2015 Micrium Inc.

OSIntCtxSw()

This function implements the interrupt level context switch which is called by `OSIntExit()` (see `os_core.c`). The pseudo code for this function is shown below (the C-like code needs to be implemented in assembly language). Refer also to Context Switching.

```
OSIntCtxSw:                                                     (1)
    OSTaskSwHook();
    OSPrioCur   = OSPrioHighRdy;
    OSTCBCurPtr = OSTCBHighRdyPtr;
    SP          = OSTCBCurPtr->StkPtr;                          (2)
    OS_CTX_RESTORE                                              (3)
    Return from Interrupt/Exception;                            (4)
```

Listing - OSIntCtxSw() Pseudo Code

(1) `OSIntCtxSw()` is called by `OSIntExit()` at the end of all nested ISRs. The ISR is assumed to have saved the context of the interrupted task onto that task's stack. Also, the ISR is assumed to have saved the new top-of-stack of the interrupted task into the `OS_TCB` of that task.

(2) The stack pointer for the new task is then retrieved from the `OS_TCB` of the new current task.

(3) `OS_CTX_RESTORE` is a macro (see `os_cpu_a.inc`) that restores the context of the CPU from the new task's stack. For our generic 32-bit CPU, `OS_CTX_RESTORE` would pop CPU registers `R13` through `R0` from the stack, in that order.

(4) The Return from Interrupt/Exception restores the Program Counter (`PC`) and the Status Register (`SR`) in a single instruction. At this point, the new task will resume execution, exactly where it was preempted.

os_cpu_a.inc

This file contains the implementation of assembly language macros that are used to simplify the implementation of `os_cpu_a.asm`. A macro replaces many assembly language instructions with a single macro invocation.

Copyright 2015 Micrium Inc.

OS_CTX_SAVE

This macro is used to save the CPU context onto the current stack. OS_CTX_SAVE needs to save the CPU registers in the same order as they are pushed in OSTaskStkInit() which is described later. OS_CTX_SAVE only saves the CPU registers that are not automatically saved by the CPU when the CPU accepts an interrupt. In other words, if the CPU automatically saves the PSW and PC onto the stack upon initiating an ISR then OS_CTX_SAVE only needs to save the remaining CPU registers.

```
OS_CTX_SAVE
    Save all the CPU registers onto the current task stack
      (in the same order as in OSTaskStkInit())
```

Listing - OS_CTX_SAVE macro Pseudo Code

Assuming our generic 32-bit CPU, OS_CTX_SAVE would be implemented as follows.

```
OS_CTX_SAVE  MACRO
    PUSH R0
    PUSH R1
    PUSH R2
    PUSH R3
    PUSH R4
    PUSH R5
    PUSH R6
    PUSH R7
    PUSH R8
    PUSH R9
    PUSH R10
    PUSH R11
    PUSH R12
    PUSH R13
ENDM
```

Listing - OS_CTX_SAVE assuming generic 32-bit CPU

OS_CTX_RESTORE

This macro is used to reverse the process done by OS_CTX_SAVE. In other words, OS_CTX_RESTORE loads the CPU registers from the stack in the reverse order..

```
OS_CTX_RESTORE
    Restore all the CPU registers from the new task's stack
      (in the reverse order that they were in OSTaskStkInit())
```

Listing - OS_CTX_SAVE macro Pseudo Code

Assuming our generic 32-bit CPU, `OS_CTX_RESTORE` would be implemented as follows.

```
OS_CTX_RESTORE  MACRO
    POP R13
    POP R12
    POP R11
    POP R10
    POP R9
    POP R8
    POP R7
    POP R6
    POP R5
    POP R4
    POP R3
    POP R2
    POP R1
    POP R0
ENDM
```

Listing - OS_CTX_RESTORE assuming generic 32-bit CPU

OS_ISR_ENTER

This macro allows you to simplify your assembly language ISRs. `OS_ISR_ENTER` is basically the first line of code you would add to the ISR. The pseudo code for `OS_ISR_ENTER` is shown below.

```
OS_ISR_ENTER
     OS_CTX_SAVE
     OSIntNestingCtr++;
     if (OSIntNestingCtr == 1) {
         OSTCBCurPtr->StkPtr = SP;
     }
```

Listing - OS_ISR_ENTER macro Pseudo Code

Assuming our generic 32-bit CPU, `OS_ISR_ENTER` would be implemented as follows. You should note that the C-like code would actually be implemented in assembly language.

Copyright 2015 Micrium Inc.

```
OS_ISR_ENTER  MACRO
    PUSH R0
    PUSH R1
    PUSH R2
    PUSH R3
    PUSH R4
    PUSH R5
    PUSH R6
    PUSH R7
    PUSH R8
    PUSH R9
    PUSH R10
    PUSH R11
    PUSH R12
    PUSH R13
    OSIntNestingCtr++;
    if (OSIntNestingCtr == 1) {
        OSTCBCurPtr->StkPtr = SP;
    }
ENDM
```

Listing - OS_ISR_ENTER assuming generic 32-bit CPU

OS_ISR_EXIT

This macro allows you to simplify your assembly language ISRs. `OS_ISR_EXIT` is basically the last line of code you would add to the ISR. The pseudo code for `OS_ISR_EXIT` is shown below.

```
OS_ISR_EXIT
    OSIntExit();
    OS_CTX_RESTORE
    Return from Interrupt/Exception
```

Listing - OS_ISR_EXIT macro Pseudo Code

Assuming our generic 32-bit CPU, `OS_ISR_EXIT` would be implemented as follows. You should note that the C-like code would actually be implemented in assembly language.

Copyright 2015 Micrium Inc.

```
OS_ISR_EXIT  MACRO
    OSIntExit();
    POP R13
    POP R12
    POP R11
    POP R10
    POP R9
    POP R8
    POP R7
    POP R6
    POP R5
    POP R4
    POP R3
    POP R2
    POP R1
    POP R0
    Return from Interrupt/Exception;
ENDM
```

Listing - OS_ISR_EXIT assuming generic 32-bit CPU

Board Support Package

About BSPs

A board support package refers to code associated with the actual evaluation board or the target board used. For example, the BSP defines functions to turn LEDs on or off, reads push-button switches, initializes peripheral clocks, etc., providing nearly any functionality to multiple products/projects.

Names of typical BSP files include:

- `bsp.c`
- `bsp.h`
- `bsp_os.c`
- `bsp_os.h`
- `bsp_os_a.a`
- `cpu_bsp.c`
- `cpu_bsp.h`

All files are generally placed in a directory as follows:

```
\Micrium\Software\EvalBoards\<manufacturer>\<board_name>\BSP\
```

Here, <manufacturer> is the name of the evaluation board or target board manufacturer and <board_name> is the name of the evaluation or target board. BSPs are typically written in C and are generally portable.

bsp.c and bsp.h

These files normally contain functions and their definitions such as:

BSP_Init()

This function is called by application code to initialize the BSP functionality. `BSP_Init()` could initialize I/O ports, setup timers, serial ports, SPI ports and so on.

cpu_bsp.c and cpu_bsp.h

These files normally contain functions and their definitions to initialize the time stamping feature needed by µC/CPU.

CPU_TS_TmrInit()

`CPU_TS_TmrInit()` is a function that µC/CPU expects to have in order to initialize the timer that will be used for timestamps. Note that a skeleton of `CPU_TS_TmrInit()` is can be found in the template file `\Micrium\Software\uC-CPU\BSP\Template\cpu_bsp.c`.

CPU_TS_TmrRd()

`CPU_TS_TmrRd()` is responsible for reading the value of a 16-, 32- or 64-bit free-running timer. `CPU_TS_TmrRd()` returns a `CPU_TS_TMR` data type which can be configured to be either a `CPU_TS32` or a `CPU_TS64`. If a 16-bit timer is used, the implementer of this function must accumulate 16-bit values into a 32-bit accumulator in order to always have 32-bit timestamps. The timer used for timestamps must count up (0, 1, 2, 3, ...) and must rollover automatically when the maximum count for the resolution of the timer is reached. In other words, a 16-bit counter should go from `0x0000` to `0xFFFF` and then roll back to 0x0000. Note that a skeleton of `CPU_TS_TmrRd()` is can be found in the template file `\Micrium\Software\uC-CPU\BSP\Template\cpu_bsp.c`.

bsp_os.c, bsp_os_a.asm & bsp_os.h

These are discussed in µC/OS-III Port.

Copyright 2015 Micrium Inc.

Testing a Port

Testing a port is fairly easy to do if you have the proper tools. Here we will assume that you have access to a good debugging tool, preferably built around an Integrated Development Environment (IDE). The debugger should allow you to load your code into your target, single step through your code, set breakpoint, examine CPU registers, look at memory contents and more.

Creating a Simple Test Project

At this point, you should have the µC/CPU and µC/OS-III port file created. To test the port files, you need to create a simple project. When you download and unzip the µC/OS-III source code you will have a directory structure similar to what is shown on the next page. When you see '`My`' in front of a directory name it means a directory you will need to create for your tests of the port.

As you will see, you can start with the template files provided with µC/CPU and µC/OS-III with little or no modifications.

Copyright 2015 Micrium Inc.

```
\Micrium
  \Software
    \EvalBoards
      \MyBoardManufacturer
        \MyBoardName
          \MyToolsName
            \MyBSP
               bsp.c
               bsp.h
               bsp_os.c                      <- See the section µC/OS-III port
               bsp_os_a.asm (optional)       <- See the section µC/OS-III port
               bsp_os.h                      <- See the section µC/OS-III port
               cpu_bsp.c
               cpu_bsp.h
            \MyTest           (1)
              app.c           (2)
              includes.h      (3)
              app_cfg.h       (4)
              cpu_cfg.h       (5)    <- Copied from \Micrium\Software\uC-CPU\Cfg\Template
              lib_cfg.h              <- Copied from \Micrium\Software\uC-LIB\Cfg\Template
              os_app_hooks.c         <- Copied from \Micrium\Software\uCOS-III\Cfg\Template
              os_app_hooks.h         <- Copied from \Micrium\Software\uCOS-III\Cfg\Template
              os_cfg.h        (6)    <- Copied from \Micrium\Software\uCOS-III\Cfg\Template
              os_cfg_app.h           <- Copied from \Micrium\Software\uCOS-III\Cfg\Template
              os_type.h              <- Copied from \Micrium\Software\uCOS-III\Source
\Micrium
  \Software
    \uC-CPU
      cpu_core.c
      cpu_core.h
      cpu_def.h
      \MyCPUName
        \MyToolsName
          cpu.h                      <- See the section µC/CPU
          cpu_a.asm                  <- See the section µC/CPU
\Micrium
  \Software
    \uC-LIB
      lib_ascii.c
      lib_ascii.h
      lib_def.h
      lib_math.c
      lib_math.h
      lib_mem.c
      lib_mem.h
      lib_str.c
      lib_str.h
\Micrium
  \Software
    \uC/OS-III
      \Cfg
        os_app_hooks.c
        os_app_hooks.h
        os_cfg.h
        os_cfg_app.h
      \Ports
        \MyCPUName
          \MyToolsName
            os_cpu.h                 <- See the section µC/OS-III port
            os_cpu_a.asm             <- See the section µC/OS-III port
            os_cpu_a.inc             <- See the section µC/OS-III port
            os_cpu_c.c         (7)   <- See the section µC/OS-III port
      \Source
        os.h
        os_cfg_app.c
        os_core.c
        os_dbg.c
```

```
        os_flag.c
        os_int.c
        os_mem.c
        os_msg.c
        os_mutex.c
        os_pend_multi.c
        os_prio.c
        os_q.c
        os_sem.c
        os_stat.c
        os_task.c
        os_tick.c
        os_time.c
        os_tmr.c
        os_type.h
        os_var.c
```

Listing - Test Code Directory Structure

(1) `MyTest` is the name of the directory that will contain the project source files.

(2) `app.c` is the test file that contains `main()` and should look as shown below.

```
/* app.c */

          #include  “includes.h”

          void  main (void)
          {
              OS_ERR  err;

              OSInit(&err);
              OSStart(&err);
          }
```

Listing - main() Example

(3) `includes.h` is a master include file that `app.c` and other files assume. The contents of this file should be as shown below. Note the use of `includes.h` is deprecated. Your application should directly `#include` only the needed headers.

```
/* includes.h */

          #include  <stdarg.h>
          #include  <stdio.h>
          #include  <stdlib.h>
          #include  <math.h>

          #include  <cpu.h>
          #include  <lib_def.h>
          #include  <lib_ascii.h>
          #include  <lib_math.h>
          #include  <lib_mem.h>
          #include  <lib_str.h>

          #include  <app_cfg.h>
          #include  <os_cfg_app.h>
          #include  <os_bsp.h>
          #include  <bsp.h>

          #include  <os.h>
```

Listing - Deprecated includes.h example

(4) `app_cfg.h` must exist but should be an 'empty' file for now.

(5) The remaining files are copied from the directories shown in the listing at the top of this page, and should not be changed at this point.

(6) You need to edit `os_cfg.h` and set the following `#define`s to the values shown below:

```
OS_CFG_ISR_POST_DEFERRED_EN  DEF_DISABLED
OS_CFG_PRIO_MAX                       32u
OS_CFG_STAT_TASK_EN          DEF_DISABLED
OS_CFG_TMR_EN                DEF_DISABLED
OS_CFG_SCHED_ROUNDROBIN_EN   DEF_DISABLED
```

This is done to ensure that we only have two tasks in the test application, `OS_IdleTask()` and `OS_TickTask()`.

For this test, you need to add *one* line of code in `OSIdleTaskHook()` as shown below. Once we verify `OSInit()`, `OSTaskStkInit()`, `OSCtxSw()`, `OS_CTX_SAVE` and `OS_CTX_RESTORE`, we'll remove this code:

Copyright 2015 Micrium Inc.

```
void  OSIdleTaskHook (void)
          {
              OSTimeTick();
          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppIdleTaskHookPtr != (OS_APP_HOOK_VOID)0) {
                  (*OS_AppIdleTaskHookPtr)();
              }
          #endif
          }
```

Listing - Testing the application with OSIdleTaskHook()

Verifying Task Context Switches

In this section, we will verify the proper operation of the following functions/macros:

```
OSInit()             (os_core.c)
OSStartHighRdy()     (os_cpu_a.asm)
OSTaskStkInit()      (os_cpu_c.c)
OSCtxSw()            (os_cpu_a.asm)
OS_CTX_SAVE          (os_cpu_a.inc)
OS_CTX_RESTORE       (os_cpu_a.inc)
CPU_SR_Save()        (cpu_a.asm)
CPU_SR_Restore()     (cpu_a.asm)
```

Our first test is to verify that µC/OS-III gets properly initialized and that the code in `OSTaskStkInit()` properly initializes a task's stack.

Recall that our application consist of `app.c` which contains the code shown below:

```
void  main (void)
          {
              OS_ERR  err;

              OSInit(&err);
              OSStart(&err);                            <- Set a BREAKPOINT here!
          }
```

STEP 1

You now need to build and download this project to your target. Building is obviously highly toolchain specific. Of course, if you encounter errors during the build, you will need to resolve those before being able to move to the next step.

Once all build errors have been resolved, you need to download the target code onto the

evaluation board you selected for the tests.

STEP 2

You then need to set a breakpoint at the `OSStart()` line. In other words, have your target stop AFTER executing `OSInit()`. You should then examine the contents of '`err`' and confirm that it has the value `OS_ERR_NONE` (or, `0`). If you get anything other than `OS_ERR_NONE`, the error code will tell you where the problem is (see section A-20).

STEP 3

If `err` is `OS_ERR_NONE` then you can 'Step Into' `OSStart()` (file `os_core.c`). You should see the following code:

```
void  OSStart (OS_ERR  *p_err)
          {
          #ifdef OS_SAFETY_CRITICAL
              if (p_err == (OS_ERR *)0) {
                  OS_SAFETY_CRITICAL_EXCEPTION();
                  return;
              }
          #endif

              if (OSRunning == OS_STATE_OS_STOPPED) {
                  OSPrioHighRdy   = OS_PrioGetHighest();                      (1)
                  OSPrioCur       = OSPrioHighRdy;
                  OSTCBHighRdyPtr = OSRdyList[OSPrioHighRdy].HeadPtr;         (2)
                  OSTCBCurPtr     = OSTCBHighRdyPtr;
                  OSRunning       = OS_STATE_OS_RUNNING;
                  OSStartHighRdy();                                           (3)
                 *p_err           = OS_ERR_FATAL_RETURN;
              } else {
                 *p_err           = OS_ERR_OS_RUNNING;
              }
          }
```

STEP 4

Step into the code and stop just before executing `OSStartHighRdy()`. You should confirm that `OSPrioCur` is the same value as `OS_CFG_TICK_TASK_PRIO` (see `os_cfg_app.h`) and that `OSTCBHighRdyPtr` point at `OSTickTaskTCB`. In other words, the highest priority task should be the tick task because we should only have two task created after `OSInit()` and the tick task always has a higher priority than the idle task.

Copyright 2015 Micrium Inc.

STEP 5

Now, ‘Step Into’ `OSStartHighRdy()` (file `os_cpu_a.asm`). You should see the assembly language shown below.

```
OSStartHighRdy:
             OSTaskSwHook();
             SP = OSTCBHighRdyPtr->StkPtr;
             OS_CTX_RESTORE
             Return from Interrupt/Exception;
```

You can ‘Step Over’ `OSTaskSwHook()` and the code to load the stack pointer. However, you should set a breakpoint at the ‘Return from Interrupt/Exception’ instruction. Once you executed the `OS_CTX_RESTORE` macro, you should look at the CPU registers and confirm that they all have their expected value (`0x12121212` for `R12`, `0x05050505` for `R5`, etc.). If not then something is not quite right with either `OSTaskStkInit()` or the `OS_CTX_RESTORE` macro. Basically, `OSTaskStkInit()` sets up the stack and `OS_CTX_RESTORE` sets up the registers based on what’s on the stack.

STEP 6

If the CPU registers appear to have their proper value then you can ‘Single Step’ and execute the ‘Return from Interrupt/Exception’ instruction. If all is well, you should be looking at the `OS_TickTask()` code which should look something like this:

```
void  OS_TickTask (void  *p_arg)
         {
             OS_ERR  err;
             CPU_TS  ts;


             p_arg = p_arg;
             while (DEF_ON) {
                 (void)OSTaskSemPend((OS_TICK  )0,
                                     (OS_OPT   )OS_OPT_PEND_BLOCKING,
                                     (CPU_TS  *)&ts,
                                     (OS_ERR  *)&err);
                 if (err == OS_ERR_NONE) {                        <- Set a BREAKPOINT here!
                     if (OSRunning == OS_STATE_OS_RUNNING) {
                         OS_TickListUpdate();
                     }
                 }
             }
         }
```

Copyright 2015 Micrium Inc.

If the debugger doesn't show you this code then it's possible that the PC and PSW are not properly setup on the task stack by OSTaskStkInit().

If you end up in OS_TickTask() your code for OSTaskStkInit() and the macro OS_CTX_RESTORE is correct.

You should now set a breakpoint on the line following OSTaskSemPend().

STEP 7

You need to set another breakpoint in OSCtxSw() as shown below.

```
OSCtxSw:                                                    <- Set a BREAKPOINT here!
              OS_CTX_SAVE
              OSTCBCurPtr->StkPtr = SP;
              OSTaskSwHook();
              OSPrioCur   = OSPrioHighRdy;
              OSTCBCurPtr = OSTCBHighRdyPtr;
              SP          = OSTCBCurPtr->StkPtr;
              OS_CTX_RESTORE
              Return from Interrupt/Exception;
```

You can now run the code at full speed. Because of the breakpoint in OSCtxSw(), the debugger should stop and show you the code for OSCtxSw().

Basically, what's happening here is that OS_TickTask() will be waiting for the tick ISR to signal the task that a tick has expired. Since we haven't setup the tick interrupt (not yet anyway), OS_TickTask() would never get to execute. However, I had you modify the idle task hook to simulate signaling the tick task so µC/OS-III will eventually switch back to this code. In the meantime, µC/OS-III will switch to the next task that's ready-to-run. This happens to be the idle task. We'll be following the code until we get to OS_IdleTask().

STEP 8

You can 'Step Over' OS_CTX_SAVE and verify that the stack (pointed to by SP) contains the value of the CPU registers saved in the same order as they are in OSTaskStkInit(). In fact, you can verify this when context switches back out of the idle task in just a few more steps.

STEP 9

'Step Into' the code one more time and verify that the SP was saved in OSTickTaskTCB.StkPtr.

Copyright 2015 Micrium Inc.

STEP 10

'Step Into' the code and stop just before executing the 'Return from Interrupt/Exception' instruction. At this point, the CPU registers should contain the proper register values (similar to what we had when we restored the CPU registers for `OSTickTask()` (but this time it's for `OS_IdleTask()`).

STEP 11

'Step Into' the return from interrupt/exception instruction and the CPU should now jump into the idle task (`os_core.c`) as shown below. You should then set a breakpoint as shown.

```
void  OS_IdleTask (void  *p_arg)
          {
              CPU_SR_ALLOC();


              p_arg = p_arg;
              while (DEF_ON) {
                  CPU_CRITICAL_ENTER();                                   <- Set a BREAKPOINT here!
                  OSIdleTaskCtr++;
          #if OS_CFG_STAT_TASK_EN > 0u
                  OSStatTaskCtr++;
          #endif
                  CPU_CRITICAL_EXIT();

                 <strong> OSIdleTaskHook();</strong>
              }
          }
```

STEP 12

'Step Into' the idle task and then, 'Step Into' `OSIdleTaskHook()`. Recall that I had you modify the idle task hook as shown below. What we're doing here is simulate the occurrence of the tick interrupt.

Copyright 2015 Micrium Inc.

```
void  OSIdleTaskHook (void)
          {
              <strong>OSTimeTick();</strong>
          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppIdleTaskHookPtr != (OS_APP_HOOK_VOID)0) {
                  (*OS_AppIdleTaskHookPtr)();
              }
          #endif
          }
```

Listing - $paramCaptionText

STEP 13

Have your debugger run the code at full speed. You should actually hit the breakpoint in `OSCtxSw()` as shown below. What happened here is that µC/OS-III signaled the tick task and since the tick task is more important than the idle task, µC/OS-III is switching back to the tick task.

```
OSCtxSw:                                                <- BREAKPOINT here.
              OS_CTX_SAVE
              OSTCBCurPtr->StkPtr = SP;
              OSTaskSwHook();
              OSPrioCur   = OSPrioHighRdy;
              OSTCBCurPtr = OSTCBHighRdyPtr;
              SP          = OSTCBCurPtr->StkPtr;
              OS_CTX_RESTORE
              Return from Interrupt/Exception;
```

Listing - $paramCaptionText

STEP 14

You can run the target at full speed and the debugger should bring you back at the breakpoint in `OS_TickTask()`.

If you were to repeatedly run the target at full speed, your debugger should now stop at the following places:

```
OSCtxSw()
OS_IdleTask()
OSCtxSw()
OS_TickTask()
OSCtxSw()
OS_IdleTask()
OSCtxSw()
```

Copyright 2015 Micrium Inc.

```
etc.
```

Verifying Interrupt Context Switches

In this section, we will verify the proper operation of the following functions/macros:

```
BSP_OS_TickISR()     (bsp_os.c, bsp_os_a.asm, os_cpu_c.c or os_cpu_a.asm)
OSIntCtxSw()         (os_cpu_a.asm)
OS_ISR_ENTER         (os_cpu_a.inc)
OS_ISR_EXIT          (os_cpu_a.inc)
CPU_INT_EN()         (cpu.h)
CPU_IntEn()          (cpu_a.asm)
```

You can now remove the code we added in `OSIdleTaskHook()`. The code should now be as shown below.

```
void  OSIdleTaskHook (void)
          {
          #if OS_CFG_APP_HOOKS_EN > 0u
              if (OS_AppIdleTaskHookPtr != (OS_APP_HOOK_VOID)0) {
                  (*OS_AppIdleTaskHookPtr)();
              }
          #endif
          }
```

You should now setup the tick interrupt in `main()` (`app.c`) as shown below.

```
/* app.c */

          #include  “includes.h”

          void  main (void)
          {
              OS_ERR  err;

              OSInit(&err);
              /* (1) Install interrupt vector for OSTickISR()                          */
              /* (2) Initialize the tick timer to generate interrupts every millisecond */
              CPU_INT_EN();       /* (3) Enable interrupts                             */
              OSStart(&err);
          }
```

(1) You need to setup the interrupt vector for the tick ISR. Where this is done greatly depends on the CPU architecture. On some processors, you would simply insert a pointer to `BSP_OS_TickISR()` in the interrupt vector table while on others, you would need to call a function to install the vector in a RAM table.

Copyright 2015 Micrium Inc.

(2) You can setup the timer you will use to generate interrupts here. You need to make sure that the interrupt will not occur immediately but instead 1 millisecond after the timer is initialized. You may recall that I told you to always initialize the tick interrupt from the first task that executes when we start multitasking. However, since we are testing the port, it's safe to initialize the timer here since we have control over when the first interrupt will actually occur.

(3) This macro is used to enable global CPU interrupts. It's assumed that the startup code runs with interrupts disabled and thus, those need to be explicitly enabled.

At this point, you need to remove all breakpoints you inserted to test the task level context switch code and insert the following breakpoints. You should note that the C-like code should actually be replaced with assembly language instructions for your processor.

```
OSIntCtxSw:                                                 <- Set BREAKPOINT here!
              OSTaskSwHook();
              OSPrioCur   = OSPrioHighRdy;
              OSTCBCurPtr = OSTCBHighRdyPtr;
              SP          = OSTCBCurPtr->StkPtr;
              OS_CTX_RESTORE
              Return from Interrupt/Exception;


          BSP_OS_TickISR:                                                <- Set BREAKPOINT here!
              OS_ISR_ENTER
              Clear tick interrupt;
              OSTimeTick();
              OS_ISR_EXIT
```

STEP 1

Reset the CPU and run the code until you hit the first breakpoint. If you properly initialized the tick timer then you should be looking at the `BSP_OS_TickISR()` code. If not, you need to determine why you are not getting the tick interrupt.

If the tick interrupt is properly setup then you should verify that is pointing at `OSIdleTaskTCB` since your application should have been looping around the idle task until the tick interrupt occurred.

Copyright 2015 Micrium Inc.

STEP 2

You should step into the `BSP_OS_TickISR()` code and verify that `OS_ISR_ENTER` increments `OSIntNestingCtr` (you should be able to look at that variable with the debugger and notice that it should have a value of `1`). Also, you should verify that the current `SP` is saved in `OSTCBCurPtr->StkPtr` (it should be the same as `OSIdleTaskTCB.StkPtr`).

STEP 3

You should now step through the code that clears the tick interrupt and verify that it's doing the proper thing.

STEP 4

You can now 'Step Over' the call to `OSTimeTick()`. `OSTimeTick()` basically signals the tick task and thus makes it ready-to-run. Instead of returning from interrupt from `BSP_OS_TickISR()`, µC/OS-III will instead exit through `OSIntCtxSw()` because the tick ISR has a higher priority than the idle task.

STEP 5

You should now 'Step Into' the code for `OS_ISR_EXIT` and 'Step Over' `OSIntExit()` (`os_core.c`). `OSIntExit()` should not return to its caller but instead, call `OSIntCtxSw()` (`os_cpu_a.asm`) as shown below. At this point, `OSTCBHighRdyPtr` should point at `OSTickTaskTCB`.

```
OSIntCtxSw:
              OSTaskSwHook();
              OSPrioCur   = OSPrioHighRdy;
              OSTCBCurPtr = OSTCBHighRdyPtr;
              SP          = OSTCBCurPtr->StkPtr;
              OS_CTX_RESTORE
              Return from Interrupt/Exception;
```

STEP 6

Before going any further in the code, you should set a breakpoint in `OS_TickTask()` (`os_tick.c`) as shown below.

```
void OS_TickTask (void *p_arg)
```

```
{
    OS_ERR err;
    CPU_TS ts;

    p_arg = p_arg;
    while (DEF_ON) {
        (void)OSTaskSemPend((OS_TICK )0,                  <- Set BREAKPOINT here!
                            (OS_OPT  )OS_OPT_PEND_BLOCKING,
                            (CPU_TS *)&ts,
                            (OS_ERR *)&err);
        if (err == OS_ERR_NONE) {
            OS_CRITICAL_ENTER();
            OSTickCtr++;
            OS_CRITICAL_EXIT();
            ts_delta_dly = OS_TickListUpdateDly();
            ts_delta_timeout = OS_TickListUpdateTimeout();
            ts_delta = ts_delta_dly + ts_delta_timeout;
            if (OSTickTaskTimeMax < ts_delta) {
                OSTickTaskTimeMax = ts_delta;
            }
        }
    }
}
```

STEP 7

You should then go back to `os_cpu_a.asm` and ‘Step Into’ the code for `OS_CTX_RESTORE` and then execute the Return from Interrupt/Exception instruction.

This should cause the code to context switch into `OS_TickTask()`. In fact, you will be in the context of `OS_TickTask()` but you will not be in the `OS_TickTask()` code itself. This is because µC/OS-III is actually returning to the point where it invoked the scheduler to switch to the idle task. µC/OS-III is simply returning to that point. You can step through code to see the path µC/OS-III is taking. However, this corresponds to quite a few lines of code. It’s probably simpler to simply run the CPU at full speed and have the debugger stop when you hit the breakpoint in `OS_TickTask()`.

If you were to repeatedly run the target at full speed, your debugger should now stop at the following breakpoints:

- `OSTickISR()`
- `OSIntCtxSw()`
- `OS_TickTask()`
- `BSP_OS_TickISR()`

- `OSIntCtxSw()`
- etc.

At this point, the port tests are complete. You should be able to use the µC/OS-III port in your target application.

Copyright 2015 Micrium Inc.

Run-Time Statistics

µC/OS-III performs substantial run-time statistics that can be displayed by kernel-aware debuggers and/or µC/Probe. Specifically, it is possible to ascertain the total number of context switches, maximum interrupt disable time, maximum scheduler lock time, CPU usage, stack space used on a per-task basis, the RAM used by µC/OS-III, and much more.

No other real-time kernel provides as much run-time information as µC/OS-III. This information is quite useful during debugging as it provides a sense of how well an application is running and the resources being used.

µC/OS-III also provides information about the configuration of the system. Specifically, the amount of RAM used by µC/OS-III, including all internal variables and task stacks.

The µC/OS-III variables described in this chapter should be displayed and never changed.

Copyright 2015 Micrium Inc.

General Statistics Run-Time

The following is a list of µC/OS-III variables that are not associated to any specific task.

Interrupts

`OSIntNestingCtr`

This variable contains the interrupt nesting level. 1 means servicing the first level of interrupt nesting, 2 means the interrupt was interrupted by another interrupt, etc.

`OSIntDisTimeMax`

This variable contains the maximum interrupt disable time (in `CPU_TS` units).

Interrupt Queue

`OSIntQNbrEntries`

This variable indicates the current number of entries in the interrupt handler queue.

`OSIntQNbrEntriesMax`

This variable contains the maximum (i.e. peak) number of entries in the interrupt handler queue. This variable is reset by `OSStatReset()`.

`OSIntQOvfCtr`

This variable shows the number of attempts to post a message from an interrupt to the interrupt handler queue, and there was not enough room to place the post call. In other words, how many times an interrupt was not being able to be serviced by its corresponding task. This value should always be 0 if the interrupt handler queue is sized large enough. If the value is non-zero, you should increase the size of the interrupt handler queue. A non-zero value may also indicate that the processor is not fast enough.

`OSIntQTaskTimeMax`

This variable contains the maximum execution time of the Interrupt Queue Handler Task

Copyright 2015 Micrium Inc.

(in CPU_TS units). The total time also includes the time of any ISR that occurred while the Interrupt Handler task was running.

Number of Kernel Objects

OSFlagQty

This variable indicates the number of event flag groups created. This variable is declared only if OS_CFG_FLAG_EN is set to DEF_ENABLED in os_cfg.h.

OSMemQty

This variable indicates the number of fixed-sized memory partitions created by the application. This variable is declared only if OS_CFG_MEM_EN is set to DEF_ENABLED in os_cfg.h.

OSMutexQty

This variable indicates the number of mutual exclusion semaphores created by the application. This variable is declared only if OS_CFG_MUTEX_EN is set to DEF_ENABLED in os_cfg.h.

OSSemQty

This variable indicates the number of semaphores created by your application. This variable is declared only if OS_CFG_SEM_EN is set to DEF_ENABLED in os_cfg.h.

OSTaskQty

The variable contains the total number of tasks created in the application.

OSTmrQty

This variable indicates the number of timers created by the application. It is declared only if OS_CFG_TMR_EN is set to DEF_ENABLED in os_cfg.h.

Message Pool

OSMsgPool.NbrFree

The variable indicates the number of free OS_MSGs in the message pool. This number should never be zero since that indicate that the application is no longer able to send messages. This variable is declared only if OS_CFG_Q_EN is set to DEF_ENABLED, or OS_CFG_TASK_Q_EN is set to DEF_ENABLED in os_cfg.h.

OSMsgPool.NbrUsed

This variable indicates the number of OS_MSGs currently used by the application. This variable is declared only if OS_CFG_Q_EN is set to DEF_ENABLED, or OS_CFG_TASK_Q_EN is set to DEF_ENABLED in os_cfg.h.

OSMsgPool.NbrUsedMax

This variable indicates the maximum (i.e. peak) number of OS_MSGs that was ever used by the application and is reset by OSStatReset(). This variable is declared only if OS_CFG_Q_EN is set to DEF_ENABLED, or OS_CFG_TASK_Q_EN is set to DEF_ENABLED in os_cfg.h.

Ready List

OSRdyList[i].NbrEntries

These variable are used to examine how many entries there are in the ready list at each priority.

Scheduler

OSSchedLockTimeMax

This variable indicates the maximum amount of time the scheduler was locked irrespective of which task did the locking. It represents the global scheduler lock time. This value is expressed in CPU_TS units. The variable is declared only if OS_CFG_SCHED_LOCK_TIME_MEAS_EN is set to DEF_ENABLED in os_cfg.h.

OSSchedLockTimeMaxCur

This variable indicates the maximum amount of time the scheduler was locked. This value is expressed in CPU_TS units and is reset by the context switch code so that it can track the scheduler lock time on a per-task basis. This variable is declared only if OS_CFG_SCHED_LOCK_TIME_MEAS_EN is set to DEF_ENABLED in os_cfg.h.

Copyright 2015 Micrium Inc.

`OSSchedLockNestingCtr`

This variable keeps track of the nesting level of the scheduler lock.

`OSSchedRoundRobinEn`

When set to 1, this variable indicates that round robin scheduling is enabled.

Statistics Task

`OSStatTaskCPUUsage`

This variable indicates the CPU usage of the application expressed as a percentage multiplied by 100. A value of 1000 indicates that 10.00% of the CPU is used, while 90.00% of the time the CPU is idling. This variable is declared only if `OS_CFG_STAT_TASK_EN` is set to `DEF_ENABLED` in `os_cfg.h`.

`OSStatTaskCPUUsageMax`

This variable indicates the maximum CPU usage of the application expressed as a percentage multiplied by 100. A value of 1000 indicates that 10.00% of the CPU is used, while 90.00% of the time the CPU is idling. This variable is declared only if `OS_CFG_STAT_TASK_EN` is set to `DEF_ENABLED` in `os_cfg.h`. and can be reset by calling `OSStatReset()`.

`OSStatTaskCtr`

This variable contains a counter that is incremented every time the idle task infinite loop runs. This variable is declared only if `OS_CFG_STAT_TASK_EN` is set to `DEF_ENABLED` in `os_cfg.h`.

`OSStatTaskCtrMax`

This variable contains the maximum number of times the idle task loop runs in 0.1 second. This value is used to measure the CPU usage of the application. This variable is declared only if `OS_CFG_STAT_TASK_EN` is set to `DEF_ENABLED` in `os_cfg.h`.

`OSStatTaskTimeMax`

This variable contains the maximum execution time of the statistic task (in `CPU_TS` units).

Copyright 2015 Micrium Inc.

It is declared only if OS_CFG_STAT_TASK_EN is set to DEF_ENABLED in os_cfg.h. The total time also includes the time of any ISR that occurred while the statistic task was running.

Tick Task

OSTickCtr

This variable is incremented every time the tick task executes.

OSTickTaskTimeMax

This variable contains the maximum execution time of the tick task (in CPU_TS units). The total time also includes the time of any ISR that occurred while the tick task was running.

OSTickListDly.NbrEntries

This variable indicates the number of tasks currently in the list of task that are waiting for time to expire. This number is updated whenever tasks are inserted or removed from that list. This field is only available when OS_CFG_DBG_EN is set to DEF_ENABLED in os_cfg.h.

OSTickListDly.NbrUpdated

This variable indicates the number of OS_TCBs, in the list of tasks waiting for time to expire, that were last updated by OS_TickTask(). In other words, if OS_TickTask() only updated the .TickRemain of one OS_TCB then this value would be set to 1. This field is only available when OS_CFG_DBG_EN is set to DEF_ENABLED in os_cfg.h.

OSTickListTimeout.NbrEntries

This variable indicates the number of tasks currently in the list of task that are pending on an object and have specified a non-zero timeout value. This number is updated whenever tasks are inserted or removed from that list. This field is only available when OS_CFG_DBG_EN is set to DEF_ENABLED in os_cfg.h.

OSTickListTimeout.NbrUpdated

This variable indicates the number of OS_TCBs, in the list of tasks pending on an object and have specified a non-zero timeout value, that were last updated by OS_TickTask(). In other words, if OS_TickTask() updated three OS_TCBs then this value would be set to 3. This field is only available when OS_CFG_DBG_EN is set to DEF_ENABLED in os_cfg.h.

Copyright 2015 Micrium Inc.

Timer Task

`OSTmrCtr`

This variable is incremented every time the timer task executes.

`OSTmrListEntries`

This variable contains the number of timers currently in the timer list. It is declared only if `OS_CFG_TMR_EN` is set to `DEF_ENABLED` in `os_cfg.h`.

`OSTmrTaskTimeMax`

This variable contains the maximum execution time of the timer task (in `CPU_TS` units). It is declared only if `OS_CFG_TMR_EN` is set to `DEF_ENABLED` in `os_cfg.h`. The total time also includes the time of any ISR that occurred while the timer task was running.

Miscellaneous

`OSIdleTaskCtr`

This variable contains a counter that is incremented every time the idle task infinite loop runs.

`OSRunning`

When non-zero, this variable indicates that multitasking has started.

`OSTaskCtxSwCtr`

This variable accumulates the number of context switches performed by C/OS-III.

Copyright 2015 Micrium Inc.

Per-Task Statistics Run-Time

µC/OS-III maintains statistics for each task at run-time. This information is saved in the task's `OS_TCB`.

`.CPUUsage`

This variable keeps track of CPU usage of the task (multiplied by 100). For example if the task's `.CPUUsage` is 200 then the task consumes 2.00% of total CPU usage.

The variable is declared only when `OS_CFG_TASK_PROFILE_EN` is set to `DEF_ENABLED` in `os_cfg.h`.

`.CPUUsageMax`

This variable keeps track of the maximum (i.e. peak) CPU usage of the task (multiplied by 100). For example if the task's `.CPUUsageMax` is 571 then the task maximum CPU usage that the task consumed at any given time is 5.71% of total CPU usage. This variable is reset by `OSStatReset()`.

The variable is declared only when `OS_CFG_TASK_PROFILE_EN` is set to `DEF_ENABLED` in `os_cfg.h`.

`.CtxSwCtr`

This variable keeps track of the number of times a task is context switched-in. This variable should increment. If it does not increment, the task is not running. At a minimum, the counter should at least have a value of one since a task is always created ready-to-run. However, if higher priority tasks prevent the task from ever running, the value would be 0.

This variable is declared only when `OS_CFG_TASK_PROFILE_EN` is set to `DEF_ENABLED` in `os_cfg.h`.

`.IntDisTimeMax`

This variable keeps track of the maximum interrupt disable time of a task (in `CPU_TS` units). This variable shows how each task affects interrupt latency.

Copyright 2015 Micrium Inc.

The variable is declared only when CPU_CFG_INT_DIS_MEAS_EN in cpu_cfg.h is set to DEF_ENABLED .

.MsgQ.NbrEntries

This variable indicates the number of entries currently waiting in the message queue of a task. This variable is declared only when OS_CFG_TASK_Q_EN is set to DEF_ENABLED in os_cfg.h.

.MsgQ.NbrEntriesMax

This variable indicates the maximum number of entries placed in the message queue of a task. This variable is declared only when OS_CFG_TASK_Q_EN is set to DEF_ENABLED in os_cfg.h.

.MsgQ.NbrEntriesSize

This variable indicates the maximum number of entries that a task message queue is able to accept before it is full.

This variable is declared only when OS_CFG_TASK_Q_EN is set to DEF_ENABLED in os_cfg.h.

.MsgQPendTime

This variable indicates the amount of time it took for a task or an ISR to send a message to the task (in CPU_TS units).

The variable is declared only when OS_CFG_TASK_PROFILE_EN is set to DEF_ENABLED in os_cfg.h.

.MsgQPendTimeMax

This variable indicates the maximum amount of time it took for a task or an ISR to send a message to the task (in CPU_TS units).

This variable is declared only when OS_CFG_TASK_PROFILE_EN is set to DEF_ENABLED in os_cfg.h.

`.PendOn`

This variable indicates what a task is pending on if the task is in a pend state. Possible values are:

- 0 Nothing
- 1 Pending on an event flag group
- 2 Pending on the task's message queue
- 3 Pending on multiple objects
- 4 Pending on a mutual exclusion semaphore
- 5 Pending on a message queue
- 6 Pending on a semaphore
- 7 Pending on a task's semaphore

`.Prio`

This corresponds to the priority of the task. This might change at run time depending on whether or not the task owns a mutual exclusion semaphore, or the user changes the priority of the task by calling `OSTaskChangePrio()`.

`.SchedLockTimeMax`

This variable keeps track of the maximum time a task locks the scheduler (in `CPU_TS` units). This variable allows the application to see how each task affects task latency. The variable is declared only when `OS_CFG_SCHED_LOCK_TIME_MEAS_EN` is set to `DEF_ENABLED` in `os_cfg.h`.

`.SemPendTime`

This variable indicates the amount of time it took for a task or ISR to signal the task (in

Copyright 2015 Micrium Inc.

CPU_TS units).

This variable is declared only when OS_CFG_TASK_PROFILE_EN is set to DEF_ENABLED in os_cfg.h.

.SemPendTimeMax

This variable indicates the maximum amount of time it took for a task or an ISR to signal the task (in CPU_TS units).

This variable is declared only when OS_CFG_TASK_PROFILE_EN is set to DEF_ENABLED in os_cfg.h.

.State

This variable indicates the current state of a task. The possible values are:

- 0 Ready
- 1 Delayed
- 2 Pending
- 3 Pending with Timeout
- 4 Suspended
- 5 Delayed and Suspended
- 6 Pending and Suspended
- 7 Pending, Delayed and Suspended

.StkFree

This variable indicates the amount of stack space (in number of stack entries) unused by a task. This value is determined by the statistic task if OS_CFG_TASK_STAT_STK_CHK_EN is set

to DEF_ENABLED in os_cfg.h.

.StkUsed

This variable indicates the maximum stack usage (in number of stack entries) of a task. This value is determined by the statistic task if OS_CFG_TASK_STAT_STK_CHK_EN is set to DEF_ENABLED in os_cfg.h.

.TickRemain

This variable indicates the amount of time left (in clock ticks) until a task time delay expires, or the task times out waiting on a kernel object such as a semaphore, message queue, or other. This value is actually a 'relative' value based on the position of the OS_TCB in the delta list. In other words, .TickRemain does not contain the actual count left.

Kernel Object Run-Time

It is possible to examine the run-time values of certain kernel objects as described in the following topics.

Semaphores

`.NamePtr`

This is a pointer to an ASCII string used to provide a name to the semaphore. The ASCII string can have any length as long as it is NUL terminated.

`.PendList.NbrEntries`

Each semaphore contains a wait list of tasks waiting for the semaphore to be signaled. The variable represents the number of entries in the wait list.

`.Ctr`

This variable represents the current count of the semaphore.

`.TS`

This variable contains the timestamp of when the semaphore was last signaled.

Mutual Exclusion Semaphores

`.NamePtr`

This is a pointer to an ASCII string used to provide a name to the mutual exclusion semaphore. The ASCII string can have any length as long as it is NUL terminated.

`.PendList.NbrEntries`

Each mutual exclusion semaphore contains a list of tasks waiting for the semaphore to be released. The variable represents the number of entries in the wait list.

`.OwnerOriginalPrio`

Copyright 2015 Micrium Inc.

This variable holds the original priority of the task that owns the mutual exclusion semaphore.

`.OwnerTCBPtr->Prio`

Dereferencing the pointer to the `OS_TCB` of the mutual exclusion semaphore owner allows the application to determine whether a task priority was changed.

`.OwnerNestingCtr`

This variable indicates how many times the owner of the mutual exclusion semaphore requested the semaphore.

`.TS`

This variable contains the timestamp of when the mutual exclusion semaphore was last released.

Memory Partitons

`.NamePtr`

This is a pointer to an ASCII string that is used to provide a name to the memory partition. The ASCII string can have any length as long as it is `NUL` terminated.

`.BlkSize`

This variable contains the block size (in bytes) for the memory partition.

`.NbrMax`

This variable contains the maximum number of memory blocks belonging to the memory partition.

`.NbrFree`

This variable contains the number of memory blocks that are available from memory partition. The number of memory blocks in use is given by:

Copyright 2015 Micrium Inc.

```
.NbrMax - .NbrFree
```

Message Queues

`.NamePtr`

This is a pointer to an ASCII string used to provide a name to the message queue. The ASCII string can have any length, as long as it is NUL terminated.

`.PendList.NbrEntries`

Each message queue contains a wait list of tasks waiting for messages to be sent to the queue. The variable represents the number of entries in the wait list.

`.MsgQ.NbrEntries`

This variable represents the number of messages currently in the message queue.

`.MsgQ.NbrEntriesMax`

This variable represents the maximum number of messages ever placed in the message queue.

`.MsgQ.NbrEntriesSize`

This variable represents the maximum number of messages that can be placed in the message queue.

Event Flags

`.NamePtr`

This is a pointer to an ASCII string used to provide a name to the event flag group. The ASCII string can have any length, as long as it is `NUL` terminated.

`.PendList.NbrEntries`

Each event flag group contains a wait list of tasks waiting for event flags to be set or cleared. This variable represents the number of entries in the wait list.

Copyright 2015 Micrium Inc.

`.Flags`

This variable contains the current value of the event flags in an event flag group.

`.TS`

This variable contains the timestamp of when the event flag group was last posted.

os_dbg.c Static

os_dbg.c is provided in µC/OS-III as some debuggers are not able to read the values of #define constants. Specifically, os_dbg.c contains ROM variables initialized to #define constants so that users can read them with any debugger.

Below is a list of ROM variables provided in os_dbg.c, along with their descriptions. These variables use approximately 100 bytes of code space.

The application code can examine these variables and you do not need to access them in a critical region as they reside in code space and are therefore not changeable.

ROM Variable	Data Type	Value
OSDbg_DbgEn	CPU_INT08U	OS_CFG_DBG_EN

When 1, this variable indicates that ROM variables in os_dbg.c will be compiled. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_ArgChkEn	CPU_INT08U	OS_CFG_ARG_CHK_EN

When 1, this variable indicates that run-time argument checking is enabled. This means that µC/OS-III will check the validity of the values of arguments passed to functions. The feature is enabled in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_AppHooksEn	CPU_INT08U	OS_CFG_APP_HOOKS_EN

When 1, the variable indicates whether application hooks will be available to the application programmer, and the pointers listed below are declared. This value is set in os_cfg.h.

```
OS_AppTaskCreateHookPtr;
OS_AppTaskDelHookPtr;
OS_AppTaskReturnHookPtr;
OS_AppIdleTaskHookPtr;
OS_AppStatTaskHookPtr;
OS_AppTaskSwHookPtr;
```

Copyright 2015 Micrium Inc.

```
OS_AppTimeTickHookPtr;
```

ROM Variable	Data Type	Value
OSDbg_EndiannessTest	CPU_INT32U	0x12345678

This variable allows a kernel awareness debugger or µC/Probe to determine the endianness of the CPU. This is easily done by looking at the lowest address in memory where this variable is saved. If the value is 0x78 then the CPU is a little endian machine. If it's 0x12, it is a big endian machine.

ROM Variable	Data Type	Value
OSDbg_CalledFromISRChkEn	CPU_INT08U	OS_CFG_CALLED_FROM_ISR_CHK_EN

When 1, this variable indicates that µC/OS-III will perform run-time checking to see if a function that is not supposed to be called from an ISR, is called from an ISR. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_FlagEn	CPU_INT08U	OS_CFG_FLAG_EN

When 1, this variable indicates that µC/OS-III's event flag services are available to the application programmer. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_FlagDelEn	CPU_INT08U	OS_CFG_FLAG_DEL_EN

When 1, this variable indicates that the `OSFlagDel()` function is available to the application programmer. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_FlagModeClrEn	CPU_INT08U	OS_CFG_FLAG_MODE_CLR_EN

When 1, this variable indicates that you can either clear or set flags when posting and pending on event flags. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value

Copyright 2015 Micrium Inc.

OSDbg_FlagPendAbortEn	CPU_INT08U	OS_CFG_FLAG_PEND_ABORT_EN

When 1, this variable indicates that the OSFlagPendAbort() function is available to the application programmer. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_FlagGrpSize	CPU_INT16U	sizeof(OS_FLAG_GRP)

This variable indicates the memory footprint (in RAM) of an event flag group (in bytes). This data type is declared in os.h.

ROM Variable	Data Type	Value
OSDbg_FlagWidth	CPU_INT16U	sizeof(OS_FLAGS)

This variable indicates the word width (in bytes) of event flags. If event flags are declared as CPU_INT08U, this variable will be 1, if declared as a CPU_INT16U, this variable will be 2, etc. This OS_FLAGS data type is declared in os_type.h.

ROM Variable	Data Type	Value
OSDbg_IntQ	CPU_INT16U	sizeof(OS_INT_Q)

This variable indicates the size (in bytes) of the OS_INT_Q data type, which is used to queue up deferred posts. The value of this variable is zero if OS_CFG_ISR_POST_DEFERRED_EN is DEF_DISABLED in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_ISRPostDeferredEn	CPU_INT08U	OS_CFG_ISR_POST_DEFERRED_EN

When 1, this variable indicates that an ISR will defer posts to task-level code. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_MemEn	CPU_INT08U	OS_CFG_MEM_EN

When 1, this variable indicates that µC/OS-III's memory management services are available to the application. This value is set in os_cfg.h.

Copyright 2015 Micrium Inc.

ROM Variable	Data Type	Value
OSDbg_MemSize	CPU_INT16U	sizeof(OS_MEM)

This variable indicates the RAM footprint (in bytes) of a memory partition control block, OS_MEM.

ROM Variable	Data Type	Value
OSDbg_MsgEn	CPU_INT08U	OS_MSG_EN

When 1, this variable indicates that the application either enabled message queues, or task message queues, or both. This value is set in os_cfg.h by ORing the value of OS_CFG_Q_EN and OS_CFG_TASK_Q_EN.

ROM Variable	Data Type	Value
OSDbg_MsgSize	CPU_INT16U	sizeof(OS_MSG)

This variable indicates the RAM footprint (in bytes) of an OS_MSG data structure.

ROM Variable	Data Type	Value
OSDbg_MsgPoolSize	CPU_INT16U	sizeof(OS_MSG_POOL)

This variable indicates the RAM footprint (in bytes) of an OS_MSG_POOL data structure.

ROM Variable	Data Type	Value
OSDbg_MsgQSize	CPU_INT16U	sizeof(OS_MSG_Q)

This variable indicates the RAM footprint (in number of bytes) of an OS_MSG_Q data type.

ROM Variable	Data Type	Value
OSDbg_MutexEn	CPU_INT08U	OS_CFG_MUTEX_EN

When 1, this variable indicates that µC/OS-III's mutual exclusion semaphore management services are available to the application. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_MutexDelEn	CPU_INT08U	OS_CFG_MUTEX_DEL_EN

Copyright 2015 Micrium Inc.

When 1, this variable indicates that the function OSMutexDel() is available to the application. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_MutexPendAbortEn	CPU_INT08U	OS_CFG_MUTEX_PEND_ABORT_EN

When 1, the variable indicates that the function OSMutexPendAbort() is available to the application. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_MutexSize	CPU_INT16U	sizeof(OS_MUTEX)

This variable indicates the RAM footprint (in number of bytes) of an OS_MUTEX data type.

ROM Variable	Data Type	Value
OSDbg_ObjTypeChkEn	CPU_INT08U	OS_CFG_OBJ_TYPE_CHK_EN

When 1, this variable indicates that µC/OS-III will check for valid object types at run time. µC/OS-III will make sure the application is accessing a semaphore if calling OSSem???() functions, accessing a message queue when calling OSQ???() functions, etc. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_PendMultiEn	CPU_INT08U	OS_CFG_PEND_MULTI_EN

When 1, this variable indicates that µC/OS-III's service to pend on multiple objects (semaphores or message queues) is available to the application. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_PendDataSize	CPU_INT16U	sizeof(OS_PEND_DATA)

This variable indicates the RAM footprint (in bytes) of an OS_PEND_DATA data type.

ROM Variable	Data Type	Value
OSDbg_PendListSize	CPU_INT16U	sizeof(OS_PEND_LIST)

Copyright 2015 Micrium Inc.

This variable indicates the RAM footprint (in bytes) of an `OS_PEND_LIST` data type.

ROM Variable	Data Type	Value
`OSDbg_PendObjSize`	`CPU_INT16U`	`sizeof(OS_PEND_OBJ)`

This variable indicates the RAM footprint (in bytes) of an `OS_PEND_OBJ` data type.

ROM Variable	Data Type	Value
`OSDbg_PrioMax`	`CPU_INT16U`	`OS_CFG_PRIO_MAX`

This variable indicates the maximum number of priorities that the application will support.

ROM Variable	Data Type	Value
`OSDbg_PtrSize`	`CPU_INT16U`	`sizeof(void *)`

This variable indicates the size (in bytes) of a pointer.

ROM Variable	Data Type	Value
`OSDbg_QEn`	`CPU_INT08U`	`OS_CFG_Q_EN`

When 1, this variable indicates that µC/OS-III's message queue services are available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
`OSDbg_QDelEn`	`CPU_INT08U`	`OS_CFG_Q_DEL_EN`

When 1, this variable indicates that the function `OSQDel()` is available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
`OSDbg_QFlushEn`	`CPU_INT08U`	`OS_CFG_Q_FLUSH_EN`

When 1, this variable indicates that the function `OSQFlush()` is available to the application. This value is set in `os_cfg.h`.

Copyright 2015 Micrium Inc.

ROM Variable	Data Type	Value
OSDbg_QPendAbortEn	CPU_INT08U	OS_CFG_Q_PEND_ABORT_EN

When 1, this variable indicates that the function OSQPendAbort() is available to the application. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_QSize	CPU_INT16U	

This variable indicates the RAM footprint (in number of bytes) of an OS_Q data type.

ROM Variable	Data Type	Value
OSDbg_SchedRoundRobinEn	CPU_INT08U	OS_CFG_SCHED_ROUND_ROBIN_EN

When 1, this variable indicates that the µC/OS-III round-robin scheduling feature is available to the application. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_SemEn	CPU_INT08U	OS_CFG_SEM_EN

When 1, this variable indicates that µC/OS-III's semaphore management services are available to the application. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_SemDelEn	CPU_INT08U	OS_CFG_SEM_DEL_EN

When 1, this variable indicates that the function OSSemDel() is available to the application. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_SemPendAbortEn	CPU_INT08U	OS_CFG_SEM_PEND_ABORT_EN

When 1, this variable indicates that the function OSSemPendAbort() is available to the application. This value is set in os_cfg.h.

ROM Variable	Data Type	Value

OSDbg_SemSetEn	CPU_INT08U	OS_CFG_SEM_SET_EN

When 1, this variable indicates that the function OSSemSet() is available to the application. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_SemSize	CPU_INT16U	sizeof(OS_SEM)

This variable indicates the RAM footprint (in bytes) of an OS_SEM data type.

ROM Variable	Data Type	Value
OSDbg_RdyList	CPU_INT16U	sizeof(OS_RDY_LIST)

This variable indicates the RAM footprint (in bytes) of the OS_RDY_LIST data type.

ROM Variable	Data Type	Value
OSDbg_RdyListSize	CPU_INT32U	sizeof(OSRdyList)

This variable indicates the RAM footprint (in bytes) of the ready list.

ROM Variable	Data Type	Value
OSDbg_StkWidth	CPU_INT08U	sizeof(CPU_STK)

This variable indicates the word size of a stack entry (in bytes). If a stack entry is declared as CPU_INT08U, this value will be 1, if a stack entry is declared as CPU_INT16U, the value will be 2, etc.

ROM Variable	Data Type	Value
OSDbg_StatTaskEn	CPU_INT08U	OS_CFG_STAT_TASK_EN

When 1, this variable indicates that µC/OS-III's statistic task is enabled. This value is set in os_cfg.h.

ROM Variable	Data Type	Value
OSDbg_StatTaskStkChkEn	CPU_INT08U	OS_CFG_STAT_TASK_STK_CHK_EN

Copyright 2015 Micrium Inc.

When 1, this variable indicates that µC/OS-III will perform run-time stack checking by walking the stack of each task to determine the usage of each. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TaskChangePrioEn	CPU_INT08U	OS_CFG_TASK_CHANGE_PRIO_EN

When 1, this variable indicates that the function `OSTaskChangePrio()` is available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TaskDelEn	CPU_INT08U	OS_CFG_TASK_DEL_EN

When 1, this variable indicates that the function `OSTaskDel()` is available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TaskQEn	CPU_INT08U	OS_CFG_TASK_Q_EN

When 1, this variable indicates that `OSTaskQ???()` services are available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TaskQPendAbortEn	CPU_INT08U	OS_CFG_TASK_Q_PEND_ABORT_EN

When 1, this variable indicates that the function `OSTaskQPendAbort()` is available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TaskProfileEn	CPU_INT08U	OS_CFG_TASK_PROFILE_EN

When 1, this variable indicates that task profiling is enabled, and that µC/OS-III will perform run-time performance measurements on a per-task basis. Specifically, when 1, µC/OS-III will keep track of how many context switches each task makes, how long a task disables interrupts, how long a task locks the scheduler, and more. This value is set in `os_cfg.h`.

Copyright 2015 Micrium Inc.

ROM Variable	Data Type	Value
OSDbg_TaskRegTblSize	CPU_INT16U	OS_CFG_TASK_REG_TBL_SIZE

This variable indicates how many entries each task register table can accept.

ROM Variable	Data Type	Value
OSDbg_TaskSemPendAbortEn	CPU_INT08U	OS_CFG_TASK_SEM_PEND_ABORT_EN

When 1, this variable indicates that the function `OSTaskSemPendAbort()` is available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TaskSuspendEn	CPU_INT08U	OS_CFG_TASK_SUSPEND_EN

When 1, this variable indicates that the function `OSTaskSuspend()` is available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TCBSize	CPU_INT16U	sizeof(OS_TCB)

This variable indicates the RAM footprint (in bytes) of an `OS_TCB` data structure.

ROM Variable	Data Type	Value
OSDbg_TimeDlyHMSMEn	CPU_INT08U	OS_CFG_TIME_DLY_HMSM_EN

When 1, this variable indicates that the function `OSTimeDlyHMSM()` is available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TimeDlyResumeEn	CPU_INT08U	OS_CFG_TIME_DLY_RESUME_EN

When 1, this variable indicates that the function `OSTimeDlyResume()` is available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TLS_TblSize	CPU_INT16U	OS_CFG_TLS_TBL_SIZE

Copyright 2015 Micrium Inc.

Indicates the size of the `.TLS_Tbl[]` in an `OS_TCB` in number of bytes. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TmrEn	CPU_INT08U	OS_CFG_TMR_EN

When 1, this variable indicates that `OSTmr???()` services are available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TmrDelEn	CPU_INT08U	OS_CFG_TMR_DEL_EN

When 1, this variable indicates that the function `OSTmrDel()` is available to the application. This value is set in `os_cfg.h`.

ROM Variable	Data Type	Value
OSDbg_TmrSize	CPU_INT16U	sizeof(OS_TMR)

This variable indicates the RAM footprint (in bytes) of an `OS_TMR` data structure.

ROM Variable	Data Type	Value
OSDbg_VersionNbr	CPU_INT16U	OS_VERSION

This variable indicates the current version of µC/OS-III multiplied by 10000. For example version 3.02.00 will show as 30200.

ROM Variable	Data Type	Value
OSDbg_DataSize	CPU_INT32U	Size of all RAM variables

This variable indicates the RAM footprint (in bytes) of the internal µC/OS-III variables for the current configuration.

Copyright 2015 Micrium Inc.

os_cfg_app.c Static

As with `os_dbg.c`, `os_cfg_app.c` defines a number of ROM variables. These variables, however, reflect the run-time configuration of an application. Specifically, the user will be able to know the RAM footprint (in bytes) of µC/OS-III task stacks, the message pool, and more.

Below is a list of ROM variables provided in `os_app_cfg.c`, along with their descriptions. These variables represent approximately 100 bytes of code space.

Application code can examine these variables and the application does not need to access them in a critical region since they reside in code space and are therefore not changeable.

ROM Variable	Data Type	Value
`OSCfg_IdleTaskStkSizeRAM`	`CPU_INT32U`	`sizeof(OSCfg_IdleTaskStk)`

This variable indicates the RAM footprint (in bytes) of the µC/OS-III idle task stack.

ROM Variable	Data Type	Value
`OSCfg_IntQSizeRAM`	`CPU_INT32U`	`sizeof(OSCfg_IntQ)`

This variable indicates the RAM footprint (in bytes) of the µC/OS-III interrupt handler task queue.

ROM Variable	Data Type	Value
`OSCfg_IntQTaskStkSizeRAM`	`CPU_INT32U`	`sizeof(OSCfg_IntQTaskStk)`

This variable indicates the RAM footprint (in bytes) of the µC/OS-III interrupt queue handler task stack.

ROM Variable	Data Type	Value
`OSCfg_ISRStkSizeRAM`	`CPU_INT32U`	`sizeof(OSCfg_ISRStk)`

This variable indicates the RAM footprint (in bytes) of the dedicated Interrupt Service Routine (ISR) stack.

ROM Variable	Data Type	Value

Copyright 2015 Micrium Inc.

OSCfg_MsgPoolSizeRAM	CPU_INT32U	sizeof(OSCfg_MsgPool)

This variable indicates the RAM footprint (in bytes) of the message pool.

ROM Variable	Data Type	Value
OSCfg_StatTaskStkSizeRAM	CPU_INT32U	sizeof(OSCfg_StatTaskStk)

This variable indicates the RAM footprint (in bytes) of the µC/OS-III statistic task stack.

ROM Variable	Data Type	Value
OSCfg_TickTaskStkSizeRAM	CPU_INT32U	sizeof(OSCfg_TickTaskStk)

This variable indicates the RAM footprint (in bytes) of the µC/OS-III tick task stack.

ROM Variable	Data Type	Value
OSCfg_DataSizeRAM	CPU_INT32U	Total of all configuration RAM

This variable indicates the RAM footprint (in bytes) of all of the configuration variables declared in `os_cfg_app.c`.

Copyright 2015 Micrium Inc.